Enterprise JavaBeans

Third Edition

Richard Monson-Haefel

O'REILLY®

Enterprise JavaBeans

Третье издание

Ричард Монсон-Хейфел

Санкт-Петербург
2002

Ричард Монсон-Хейфел

Enterprise JavaBeans, 3-е издание

Перевод И. Васильева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Шальнев*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*А. Дорошенко*

Монсон-Хейфел Р.
Enterprise JavaBeans, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 672 с., ил.
ISBN 5-93286-041-3

Третье издание «Enterprise JavaBeans» описывает технологию построения сложных ответственных систем, основанную на объединении компонентов, моделирующих прикладные объекты и процессы. EJB берет на себя заботу об объектном постоянстве, безопасности и управлении транзакциями. Еще недавно казалось невероятным, что компоненты EJB смогут не только выполняться в любой операционной системе без какой-либо модификации, но и работать на любом корпоративном сервере EJB. В EJB 2.0 введены компоненты, управляемые сообщениями, способные с помощью службы сообщений JMS взаимодействовать с системами промежуточного уровня; предложена более развитая модель управляемого контейнером постоянства и поддерживаются сложные отношения между объектными компонентами.

В книге рассматриваются: EJB 2.0 и 1.1, сеансовые и объектные компоненты (включая новую модель CMP и язык запросов EJB QL), компоненты, управляемые сообщениями, и служба JMS, XML-дескрипторы развертывания, управление транзакциями и безопасность, взаимосвязь EJB и Java 2, Enterprise Edition. Опытные разработчики корпоративных программных продуктов знают, как сильно технология EJB изменила эту область. Данное третье издание поможет им освоить последние достижения. А для тех, кто не имеет опыта работы с EJB, автор подготовил стартовую площадку, с которой они могут начать изучение этой захватывающей технологии построения прикладных систем.

ISBN 5-93286-041-3
ISBN 0-596-00226-2 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 15.07.2002. Формат 70×100 1/16 . Печать офсетная.
Объем 42 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Заметки автора

Зимой 1997 года я консультировал один проект по электронной коммерции, использовавший Java RMI. Не удивительно, что проект провалился, потому что Java RMI не ориентирован на производительность, масштабируемость, надежность, безопасность и транзакции – на все то, что жизненно необходимо производственному окружению. Хотя исход этого проекта не является уникальным для Java RMI, я вижу, что то же самое происходит и с CORBA – время, в которое проводилась реализации этого проекта, было особенно интересным. Примерно в это же время Sun Microsystems впервые представила Enterprise JavaBeans™, и если бы Enterprise JavaBeans стал доступным немного раньше, этот проект, возможно, завершился бы успешно.

Во время работы над тем злополучным проектом я также вел колонку в «JavaReport Online» под названием «The Cutting Edge» (Срезанный угол). Колонка охватывала новые на тот момент технологии Java, такие как Java Naming and Directory Interface™ (JNDI) и JavaMail™ API. Я искал новую тему для третьего выпуска «Срезанного угла», когда обнаружил первый опубликованный черновой проект Enterprise JavaBeans версии 0.8. Первый раз я услышал об этой технологии в 1996 году, но открытая документация стала доступной только в начале 1997. Работая с CORBA, Java RMI и другими технологиями распределенных объектов, я понял, что это хорошая вещь, и сразу же начал свою статью об этой новой технологии.

Кажется, что прошла уже вечность. С тех пор как я опубликовал статью в марте 1998 года, о EJB были написаны буквально сотни статей и уже несколько книг пришло и ушло. Эта книга (уже третье издание) идет в ногу с тремя версиями спецификации EJB много лет. Сейчас, когда выходит очередная новая версия спецификации и появляется множество новых книг на эту тему, я не могу удержаться, чтобы не вспомнить те дни, когда слова «Enterprise JavaBeans» почти у всех вызывали непонимающие взгляды. И я рад, что эти времена прошли навсегда.

Что такое Enterprise JavaBeans?

Летом 1995 года, после первого представления Java, почти вся индустрия информационных технологий обратила внимание на возможности графического пользовательского интерфейса и конкурентоспособные преимущества, предлагаемые этой технологией с точки зрения переносимости и платформенной независимости. Это было интересное время. Лишь немногие из нас пробовали применять апплеты на стороне сервера. В действительности мы тратили половину нашего времени на программирование, а другую – на то, чтобы убедить руководство, что Java – это не прихоть моды.

Сегодня фокус значительно расширился: язык Java получил признание как отличная платформа для создания корпоративных решений, особенно при разработке распределенных серверных приложений. Такое смещение акцентов произошло благодаря становлению Java как универсального языка для создания независимых от реализации абстракций для всех распространенных корпоративных технологий. Программный интерфейс JDBC – первый и наиболее известный пример. JDBC (Java Database Connectivity) предлагает независимый от производителя Java-интерфейс для доступа к базам данных. Эта абстракция была такой успешной, что сейчас трудно найти производителя реляционных баз данных, который не поддерживал бы JDBC. Абстракции Java значительно расширились и включают JNDI для абстрактных служб каталогов, JTA (Java Transaction API – программный интерфейс транзакций) для абстрактного доступа к менеджерам транзакций, JMS (Java Message Service – служба сообщений) для абстрактного доступа к различным продуктам, ориентированным на сообщения.

Технология Enterprise JavaBeans впервые была представлена в виде рабочей спецификации в конце 1997 года и заявила о себе как о наиболее значительной из всех корпоративных технологий Java, предлагаемых компанией Sun Microsystems. EJB обеспечивает абстракцию для мониторов компонентных транзакций (CTM), представляющих собой сплав двух технологий: традиционных мониторов обработки транзакций (TP), таких как CLCS, TUXEDO и Encina, и служб распределенных объектов, таких как CORBA, DCOM и «родной» для Java RMI. Соединив в себе все самое лучшее из обеих технологий, мониторы компонентных транзакций предлагают мощное объектно-ориентированное окружение, которое упрощает распределенную разработку и при этом автоматически управляет наиболее сложными аспектами корпоративных вычислений, такими как взаимодействие между объектами, управление транзакциями, безопасность, постоянство и параллелизм.

Enterprise JavaBeans определяет модель серверных компонентов, позволяющую разрабатывать прикладные объекты и перемещать их между контейнерами EJB, созданными разными производителями. Ком-

понент (enterprise bean) представляет собой простую программную модель, которая позволяет разработчику сконцентрироваться на стоящей перед ним конкретной проблеме. А сервер EJB отвечает за то, чтобы сделать этот компонент распределенным объектом, и за управляющие службы, такие как транзакции, постоянство, параллелизм и безопасность. Кроме создания бизнес-логики компонента разработчик задает атрибуты поведения компонента во время выполнения, используя подход, похожий на выбор свойств отображения визуальных элементов. Поведение компонента по отношению к транзакциям, постоянству и безопасности может быть задано путем выбора из списка свойств. Конечный результат состоит в том, что EJB значительно облегчает создание систем распределенных компонентов, управляемых мощными транзакционными средами. Для разработчиков и корпоративных информационно-технических отделов, которые борются с проблемами развертывания надежных, производительных распределенных систем на базе CORBA, DCOM и Java RMI, EJB предоставляет в качестве основы для программирования более простую и более производительную платформу.

В 1998 году была завершена спецификация Enterprise JavaBeans 1.0, которая фактически сразу же стала промышленным стандартом. Многие производители объявили о поддержке спецификации еще до ее завершения. С того времени EJB дважды улучшалась. Первый раз спецификация была обновлена в 1999 году до версии 1.1, которая была рассмотрена во втором издании этой книги. Самый последний вариант спецификации – версия 2.0 – описывается в этом, третьем издании «Enterprise JavaBeans», охватывающем также версию EJB 1.1, которая, по большей части, является подмножеством функциональных возможностей, предлагаемых EJB 2.0.

Продукты, соответствующие стандартам EJB, приходят из всех секторов информационной индустрии, в числе которых мониторы обработки транзакций (TPM), CORBA ORB, серверы приложений, реляционные и объектные базы данных, веб-серверы. Некоторые из этих продуктов базируются на собственных моделях, адаптированных под EJB.

Вкратце, обе версии – Enterprise JavaBeans 2.0 и 1.1 – представляют стандартную модель распределенных компонентов, которая значительно упрощает процесс разработки и позволяет компонентам, созданным и развернутым на сервере EJB одного производителя, с легкостью быть развернутыми на сервере другого производителя. Эта книга дает основу, необходимую для создания независимых от производителя EJB-проектов.

Для кого предназначена эта книга?

Эта книга объясняет и демонстрирует основные принципы архитектур Enterprise JavaBeans 2.0 и 1.1. Хотя EJB сильно упрощает распреде-

ленные вычисления, она достаточно сложна и требует много времени для овладения ею на уровне мастера. Книга понятно и последовательно разъясняет лежащую в основе технологию, классы и интерфейсы Java, компонентную модель и поведение компонентов во время выполнения программы. Она содержит материал, обратно совместимый с EJB 1.1, и включает отдельные замечания и главы, охватывающие наиболее значимые различия между версиями 1.1 и 2.0.

Хотя эта книга сфокусирована на основах, она не предназначена для начинающих. Enterprise JavaBeans – крайне сложная и амбициозная корпоративная технология. Применение EJB может быть достаточно простым, но для ее полного освоения и правильного понимания требуется очень много времени. Перед тем как читать эту книгу вы должны свободно овладеть языком Java и получить практический опыт создания бизнес-проектов. Опыт работы с системами распределенных объектов не требуется, но для того чтобы разобраться с примерами, приведенными в книге, необходимо определенное знание (или хотя бы понимание основ) технологии JDBC. Если вы не знакомы с языком Java, я рекомендую книгу «Learning Java» (Изучаем Java) Патрика Нимайера (Patrick Niemeyer) и Джонатана Кнудсена (Jonathan Knudsen), которая первоначально называлась «Exploring Java». Тем же, кто не знаком с JDBC, рекомендую книгу «Database Programming with JDBC and Java» (Программирование баз данных с помощью JDBC и Java) Джорджа Риза (George Reese), а тем, кому нужны прочные знания распределенных вычислений – «Java Distributed Computing» (Распределенные вычисления на Java) Джима Фарли (Jim Farley).

Структура книги

Первые три главы содержат общий ознакомительный материал, рассматривающий Enterprise JavaBeans 2.0 и 1.1 в контексте других родственных технологий и объясняющий на максимально отвлеченном уровне, как работает технология EJB и из чего состоят компоненты EJB. В главах с 4 по 13 подробно объяснено, как разрабатываются компоненты различных типов. Главы 14 и 15 могли бы считаться материалом для дополнительного изучения, если бы не транзакции (глава 14), которые существенны для всего, что происходит в корпоративных вычислениях, и стратегии дизайна (глава 15), помогающие справляться с реальными проблемами, влияющими на разработку компонентов. В главе 16 подробно описан дескриптор развертывания на базе XML, используемый в EJB 2.0 и 1.1. Глава 17 дает общий обзор пакета Java 2, Enterprise Edition (J2EE), включающего сервлеты, Java Server Pages (JSP) и EJB. Наконец, три приложения предоставляют справочную информацию, которая может быть вам полезна.

Глава 1. «Введение»

Здесь вводится определение мониторов компонентных транзакций и объясняется, каким образом они формируют технологию, лежащую в основе модели компонентов Enterprise JavaBeans.

Глава 2. «Обзор архитектуры»

Вводится понятие архитектуры компонентной модели Enterprise JavaBeans и рассматриваются различия между тремя основными типами компонентов: объектными компонентами, сеансовыми компонентами и компонентами, управляемыми сообщениями.

Глава 3. «Управление ресурсами и основные службы»

Объясняется, как EJB-совместимый сервер управляет компонентами во время выполнения.

Глава 4. «Создание вашего первого компонента»

В этой главе описываются все этапы разработки простого компонента.

Глава 5. «Клиентское представление»

Содержит подробный рассказ о том, как удаленное клиентское приложение получает доступ и использует компоненты.

Глава 6. «EJB 2.0 CMP: Основы постоянства»

Здесь объясняется, как в EJB 2.0 создать простой объектный компонент, управляемый контейнером.

Глава 7. «EJB 2.0 CMP: Отношения между объектами»

В этой главе продолжается тема, начатая в главе 6, расширяя границы вашего понимания от постоянства, управляемого контейнером, до сложных взаимоотношений между компонентами.

Глава 8. «EJB 2.0 CMP: EJB QL»

Эта глава обращается к языку запросов Enterprise JavaBeans (EJB Query Language, EJB QL), который применяется для обращения к компонентам и для поиска определенных компонентов при использовании постоянства, поддерживаемого контейнером EJB.

Глава 9. «EJB 1.1 CMP»

Описывается постоянство, управляемое контейнером в EJB 1.1, которое для обратной совместимости поддерживается и в EJB 2.0. Прочитайте эту главу, если только вам нужно поддерживать существующие EJB-приложения.

Глава 10. «Постоянство, реализуемое компонентом»

Здесь рассказывается о создании компонентов, самостоятельно реализующих постоянство (bean-managed persistence), в том числе о том, когда сохранять, загружать и удалять данные из базы данных.

Глава 11. «Соглашения между объектом и контейнером»

Эта глава описывает общий протокол взаимодействия между объектным компонентом и его контейнером во время выполнения, применяемый для реализации постоянства, поддерживаемого контейнером как в версии 2.0, так и в 1.1.

Глава 12. «Сеансовые компоненты»

В этой главе показано, как создавать сеансовые компоненты с состоянием и без состояния.

Глава 13. «Компоненты, управляемые сообщениями»

Показано, как в EJB 2.0 создавать компоненты, управляемые сообщениями.

Глава 14. «Транзакции»

Эта глава дает глубокое объяснение транзакций и описывает модель транзакций, введенную в Enterprise JavaBeans.

Глава 15. «Стратегии дизайна»

Даются некоторые основные стратегии дизайна, уменьшающие усилия, требуемые для разработки компонентов и делающие вашу систему EJB более эффективной.

Глава 16. «XML-дескрипторы развертывания»

В этой главе представлено подробное описание XML-дескрипторов развертывания, используемых в EJB 1.1 и 2.0.

Глава 17. «Java 2, Enterprise Edition»

Приводится обзор J2EE 1.3 и объясняется, как EJB 2.0 встраивается в эту новую платформу.

Приложение A. «Программный интерфейс Enterprise JavaBeans»

Данное приложение представляет собой краткий справочник по классам и интерфейсам, определенным в пакетах EJB.

Приложение B. «Диаграммы состояний и последовательностей»

Представлены диаграммы, поясняющие жизненный цикл компонентов во время выполнения.

Приложение C. «Производители EJB»

В этом приложении приводится информация о производителях серверов EJB.

Программы и версии

Эта книга описывает Enterprise JavaBeans версий 2.0 и 1.1, включая все дополнительные возможности. В ней используются особенности языка Java из платформы Java 1.2 и JDBC. Ввиду того что данная книга нацелена на создание компонентов Enterprise JavaBeans и проек-

тов, не зависящих от конкретного производителя, я старался избегать нестандартных расширений и диалектов, специфичных для определенного производителя. Для чтения этой книги достаточно любого совместимого с EJB сервера, но работа с примерами требует знания процедур установки, развертывания и управления сервером во время выполнения.

EJB 2.0 и 1.1 имеют много общего, но там, где они отличаются, главы или разделы глав, специфичные для каждой версии, выделены особо. Можете пропускать те разделы, которые вас не касаются. Если не указано обратное, исходные коды из этой книги будут работать как для EJB 2.0, так и для EJB 1.1.

Примеры, приведенные в этой книге, доступны на сайте *ftp://ftp.oreilly.com/pub/examples/java/ejb*. Примеры сгруппированы по главам.

Рабочие книги с примерами

Хотя EJB-приложения сами по себе переносимы, способы установки и запуска EJB-продуктов варьируются от производителя к производителю. По этой причине не представлялось возможным охватить все доступные EJB-продукты и был выбран радикальный, но в то же время эффективный способ, позволяющий рассмотреть эти различия, – рабочие упражнения (workbooks).

Для того чтобы помочь читателям установить примеры из книги на различные EJB-продукты, я опубликовал несколько бесплатных рабочих упражнений, которые можно использовать вместе с этой книгой для запуска примеров на конкретных коммерческих и некоммерческих серверах. Рабочие книги для конкретного продукта будут относиться к самой последней версии сервера. Так, если производитель поддерживает EJB 2.0, пример из рабочей книги будет использовать особенности EJB 2.0. С другой стороны, если производитель поддерживает только EJB 1.1, примеры будут специфичными для версии 1.1.

Я планирую опубликовать рабочие книги для нескольких различных серверов EJB, и по крайней мере две из них будут доступны в самое ближайшее время. Эти книги будут бесплатно доступны (в формате PDF) на *http://www.oreily.com/catalog/entjbeans3/* и *http://www.titanbooks.com*.

Соглашения

В книге приняты следующие типографские соглашения:

Курсив

Предназначен для имен файлов и путей, имен хостов, доменов, URL и адресов электронной почты. Курсивом также выделяются вновь вводимые термины.

`Моноширинный`

Используется в примерах программ и их фрагментов, для элементов XML, тегов, команд SQL, имен таблиц и колонок. Моноширинный шрифт применяется также для имен классов, переменных методов и для ключевых слов Java, используемых в тексте.

`Моноширинный полужирный`

Служит для выделения в некоторых примерах программ.

`Моноширинный курсив`

Указывает на то, что текст должен быть заменен. Например, в слове *`BeanName`*`PK` следует заменить *`BeanName`* конкретным именем.

Указывает на подсказки, советы, общие замечания.

Означает предупреждение или предостережение.

Компонент Enterprise JavaBeans состоит из нескольких частей. Это не один объект, а набор объектов и интерфейсов. Для ссылки на компонент как на единое целое мы будем использовать его прикладное имя, набранное основным шрифтом. Например, будем ссылаться на компонент Customer тогда, когда нам нужно обозначить компонент вообще. Если мы набираем имя моноширинным шрифтом, значит, мы явно ссылаемся на удаленный интерфейс. Таким образом, `CustomerRemote` представляет собой удаленный интерфейс, определяющий прикладные методы компонента Customer.

Комментарии и вопросы

Пожалуйста, направляйте комментарии и вопросы, касающиеся этой книги, в издательство:

O'Reilly & Associaties, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (для США и Канады)
(707) 829-0515 (международный и местный)
(707) 829-0104 (факс)

У этой книги есть веб-страница, содержащая найденные опечатки, примеры и некоторую дополнительную информацию. Ее можно найти по адресу:

http://www.orielly.com/catalog/entjbeans3/

Комментарии и технические вопросы направляйте по адресу:

bookquestions@orielly.com

За дополнительной информацией о книгах, конференциях, программных продуктах, центрах ресурсов и сети O'Reilly обращайтесь на вебсайт O'Reilly:

http://www.orielly.com

Автор поддерживает сайт для дискуссий по EJB и связанным с ней распределенным компьютерным технологиям на *http://www.jmiddleware.com*. *jMiddleware.com* предлагает последние новости по этой книге, а также подсказки по программированию, статьи и большой список ссылок на ресурсы EJB.

Благодарности

Хотя на обложку этой книги вынесено только одно имя, к ее написанию и распространению приложили усилия многие люди. Успех каждого издания этой книги зависел в основном от Майкла Лукидеса (Michael Loukides). Без его опыта, ловкости и руководства этой книги просто не было бы. Другие люди из O'Reilly & Associates, включая Рэчел Уилер (Rachel Wheeler), Роба Романо (Rob Romano), Кайла Харта (Kyle Hart) и многих других, также много сделали для успеха этой книги.

Многие технические редакторы помогли обеспечить то, что материал о сущности Enterprise JavaBeans был технически точным и правильным. Отдельно нужно упомянуть Грега Найберга (Greg Nyberg) из Object Partners, Хеманта Ханделвала (Hemant Khandelwal) из Pramati, Кайла Брауна (Kyle Brown) из IBM, Роберта Кастанеда (Robert Castaneda) из Custom Ware, Джоя Фиалли (Joy Fially) из Sun Microsystems, Анила Шарму (Anila Sharma) и Прайенка Растджи (Priyank Rastogi) из Pramanti, Сева Уйта (Seth White) из BEA, Эвана Ирланда (Evan Ireland) и Майера Тануана (Meyer Tanuan) из Subase, Дэвида Чаппела (David Chappell) из David Chappell & Associates, Джима Фарли (Jim Farley) – автора «Java Distributed Computing» (O'Reilly) и Прасада Муппиралу (Prasad Muppirala) из BORN Information Services. Они во многом способствовали обеспечению технической точности книги и вложили свой профессиональный и жизненный опыт, помогая сделать ее лучшей книгой по Enterprise JavaBeans из выпускаемых сегодня.

Я хотел бы поблагодарить всех тех, кто присылал ценные замечания по книге во время ее написания по адресу *http://orielly.techrev.org*, включая (в алфавитном порядке):

Диона Алмаера (Dion Almaer), Джима Арчера (Jim Archer), Стефана Дейвиса (Stephen Davis), Джона Де Ла Круза (John De La Cruz), Тома Гулло (Tom Gullo), Марти Харви (Marty Harvey), Хая Хонга (Hai Hoang), Мирея Куннумпурата (Meeraj Kunnumpurath), Тома Ларсона (Tom Larson), Бьерна Расмуссена (Bjarne Rassmussen), rgparker (имя неизвестно), Лари Сельцера (Larry Seltzer) и Курта Смита (Curt Smith).

Отдельные благодарности Шрираму Шринивасону (Sriram Srinivason) из BEA, Анне Томас (Anne Tomas) из Sun Microsystems, Яну Маккалиону (Ian McCallion) из IBM Hursley, Тиму Рохайли (Tim Rohaly) из jGuru.com, Джеймсу Френтрессу (James D Frentress) из корпорации ITM, Анджее Джан Тарамину (Andrzej Jan Taramina) из Accredo Systems, Марку Лою (Marc Loy), соавтору книги «Java Swing», Дону Уэйсу (Don Weiss) из Step 1, Майку Слину (Mike Slinn) из корпорации The Dialog и Кэвину Дику (Kevin Dick) из Kevin Dick & Associates. Помощь этих технических экспертов была очень важна для технической и принципиальной точности первых изданий этой книги. Кроме этого, я хотел бы поблагодарить Мегги Мескита (Maggie Mezquita), Грэга Хартсела (Greg Hartzel), Джона Клуга (John Klug) и Яна Джамсу (Jon Jamsa) из BORN Information, которые просмотрели первый вариант первого издания и составили ценные замечания.

Также хочу поблагодарить Влада Матену (Vlad Matena) и Марка Хапнера (Mark Hapner) из Sun Microsystem, главных архитекторов Enterprise JavaBeans, Линду Де Мишель (Linda DeMichiel), руководителя спецификации EJB 2.0, и Бони Кельта (Bonnie Kellett), менеджера программы J2EE, за их желание ответить на мои самые сложные вопросы. Благодарю всех участников списка рассылки EJB-INTEREST, поддерживаемого компанией Sun Microsystems за их интересные и иногда противоречивые, но всегда информативные послания за последние четыре года.

И наконец, выражаю самую искреннюю благодарность моей жене Холли за помощь и поддержку в течение трех лет мучительных исследований и написания всего того, что потребовалось для создания трех изданий этой книги. Без ее неизменной поддержки и любви эта книга не была бы завершена.

1

Введение

Эта книга об Enterprise JavaBeans 2.0 и 1.1, представляющих собой третью и вторую версии спецификации Enterprise JavaBeans. Точно так же, как платформа Java кардинально изменила наши представления о разработке программного обеспечения, Enterprise JavaBeans изменила представления о разработке надежных корпоративных программных систем. Она сочетает серверные компоненты с технологиями распределенных объектов и системами асинхронных сообщений, что значительно упрощает задачу разработки приложений. Кроме того, Enterprise JavaBeans автоматически учитывает многие требования прикладных систем, включая безопасность, поддержание пула ресурсов (resource pooling), постоянство, параллелизм и целостность транзакций.

В книге показано, как использовать Enterprise JavaBeans для создания масштабируемых, переносимых прикладных систем. Но перед тем как начать разговор о самой EJB, необходимо вкратце рассмотреть технологии, связанные с EJB, такие как модели компонентов, распределенные объекты, мониторы компонентных транзакций (Component Transaction Monitor, CTM) и системы асинхронных сообщений. Особенно важно иметь общее представление о мониторах компонентных транзакций, технологии, лежащей в основе EJB. В главах 2 и 3 мы начнем рассмотрение самой технологии EJB и увидим, как ее компоненты работают вместе. Остаток этой книги посвящен разработке компонентов для вымышленного предприятия и обсуждению сложных вопросов.

Предполагается, что читатель уже знаком с Java, а если нет, то книга «Learning Java» (Изучаем Java) Патрика Нимайера и Джона Пека будет превосходным введением в тему. Предполагается также, что вы знакомы с интерфейсом JDBC или, по крайней мере, с SQL. Если же нет – взгляните на «Database programming with JDBC and Java» (Программирование баз данных с помощью JDBC и Java) Джорджа Риза (George Reese).

Одна из наиболее важных особенностей Java – ее платформенная независимость. Со времени своего первого выпуска Java присутствует на рынке под лозунгом «написать однажды, использовать везде». Хотя эта реклама временами кажется немного неуклюжей, код, написанный на языке Java, на удивление независим от платформы. Enterprise JavaBeans не просто не зависит от платформы, она также не зависит и от реализации. Тот, кто работал с JDBC, должен немного представлять, что это значит. Интерфейс JDBC может не только работать на машинах с Windows или UNIX, он также может обращаться к реляционным базам данных от множества различных производителей (DB2, Oracle, Sybase, SQLServer и т. д.), используя различные драйверы JDBC. Нет необходимости программировать конкретную реализацию базы данных, просто замените драйвер JDBC, и будет заменена база данных. То же самое и с EJB. В идеале компонент EJB (enterprise bean) может выполняться на любом сервере приложений, реализующем спецификацию EJB.[1] Это значит, что можно разработать и развернуть прикладную систему на одном сервере, таком как Orion или Weblogic от BEA, а позже переместить ее на другой сервер EJB, такой как Pramati, Sybase EAServer, WebSphere от IBM или на проект с открытым кодом, например OpenEJB, JonAS или Jboss. Независимость от реализации означает, что прикладные компоненты не зависят от марки используемого сервера, что дает свободу выбора до, во время и после разработки и развертывания.

Подготовительный этап

Прежде чем дать более точное определение Enterprise JavaBeans, обсудим некоторые важные концепции: распределенные объекты, прикладные объекты, мониторы компонентных транзакций и системы асинхронных сообщений.

Распределенные объекты

Благодаря распределенным вычислениям прикладные системы становятся более доступными. Распределенные системы позволяют своим

[1] При условии, что и компоненты и серверы EJB соответствуют этой спецификации и при разработке не используются никакие нестандартные расширения.

отдельным частям размещаться на разных компьютерах, возможно, расположенных в разных местах, там, где это необходимо. Другими словами, распределенные вычисления обеспечивают доступность прикладной логики и данных из удаленных мест. Клиенты, деловые партнеры и другие удаленные группы пользователей могут работать с прикладной системой в любое время и почти отовсюду. Самая последняя разработка в области распределенных вычислений – *распределенные объекты* (*distributed objects*). Благодаря технологиям распределенных объектов, таким как Java RMI, CORBA и .NET от Microsoft, объекты, работающие на одной машине, могут использоваться клиентскими приложениями, выполняющимися на других компьютерах.

Распределенные объекты возникли на основе достаточно старой модели трехуровневой архитектуры, применяемой в системах с мониторами обработки транзакций (TP), таких как CICS от IBM и TUXEDO от BEA. Эти системы разбиваются на три отдельных уровня, или слоя. В прошлом эти системы состояли из «зеленых экранов», или «тупых» (dumb) терминалов, на уровне представления (первый уровень), приложений на языках COBOL или PL/1 в качестве промежуточного уровня (второй уровень) и какой-нибудь базы данных, например DB2, в качестве источников данных (третий уровень). Появление в последние годы распределенных объектов привело к появлению новой формы трехуровневой архитектуры. Технологии распределенных объектов позволили заменить процедурные приложения второго уровня, использующие COBOL и PL/1, новыми прикладными объектами. Трехуровневая архитектура распределенных прикладных объектов может иметь изощренный графический или веб-интерфейс в качестве первого уровня, прикладные объекты на втором уровне и реляционные либо другие базы данных в качестве хранилища данных. Более сложные архитектуры часто имеют больше уровней: различные объекты размещаются на разных серверах и взаимодействуют между собой, выполняя общую задачу. Создание таких многоуровневых архитектур с помощью Enterprise JavaBeans выполняется сравнительно легко.

Серверные компоненты

Объектно-ориентированные языки, такие как Java, C++ и Smalltalk, применяются для написания программ, являющихся гибкими, расширяемыми и пригодными для повторного использования, – это три аксиомы объектно-ориентированного подхода. В прикладных системах объектно-ориентированные языки служат для ускорения разработки пользовательского интерфейса, упрощения доступа к данным и для инкапсуляции прикладной логики. Инкапсуляция прикладной логики в *прикладные объекты* – достаточно новое направление в области информационных технологий. Бизнес очень мобилен, а это значит, что бизнес-продукты, процессы и цели находятся в постоянном развитии. Если программа, моделирующая бизнес-процесс, может

быть инкапсулирована или встроена в прикладной объект, она становится гибкой, расширяемой и пригодной для повторного использования и, таким образом, может развиваться вместе с бизнесом.

Модель серверных компонентов способна задать архитектуру для разработки *распределенных прикладных объектов* (*distributed business objects*), которые сочетают в себе доступность систем распределенных объектов с изменчивостью, воплощенной в их прикладной логике. Модель серверных компонентов используется в серверах приложений промежуточного уровня, которые управляют компонентами во время выполнения и делают их доступными для удаленных клиентов. Они предоставляют основу функциональности, облегчающую разработку распределенных прикладных объектов и сборку их в прикладной проект.

Серверные компоненты также могут применяться и для моделирования других аспектов прикладных систем, таких как представление и маршрутизация. Сервлеты Java, например, являются серверными компонентами, предназначенными для создания данных в форматах HTML и XML, используемых на уровне представления трехуровневой архитектуры. Компоненты EJB 2.0, управляемые сообщениями, обсуждаемые далее, являются серверными компонентами, используемыми для приема и обработки асинхронных сообщений.

Серверные компоненты, как и другие компоненты, могут продаваться и покупаться в качестве независимых частей исполняемых программных продуктов. Они соответствуют стандартной модели компонентов и могут без непосредственной модификации работать на сервере, поддерживающем эту компонентную модель. Модели серверных компонентов часто поддерживают программирование, основанное на атрибутах (attribute-based programming), позволяющее изменять поведение компонента во время выполнения при его развертывании, без необходимости изменения его программного кода. В зависимости от используемой компонентной модели администратор сервера может задать поведение объекта, относящееся к транзакциям, безопасности и даже постоянству, задав для этих атрибутов определенные значения.

По мере развития корпоративных служб, продуктов и алгоритмов работы серверные компоненты могут перекомпоновываться, изменяться и расширяться так, чтобы отражать изменения, происходящие в бизнес-системе. Представим себе прикладную систему в виде набора серверных компонентов, моделирующих такие понятия, как клиенты, продукты, заказы и товарные склады. Каждый компонент похож на блок из набора Lego, который может комбинироваться с другими компонентами для построения прикладных проектов. Продукты могут храниться на складах или отправляться клиентам, клиенты могут делать заказы и покупать продукты. Компоненты можно собирать вместе, раскладывать их по отдельности, использовать в различных комби-

нациях и изменять их определения. Прикладная система, основанная на серверных компонентах, обладает гибкостью благодаря разделению ее на объекты и доступностью благодаря тому, что компоненты могут быть рассредоточены.

Мониторы компонентных транзакций

Не так давно возникло новое поколение программных продуктов, называемых *серверами приложений* (*application server*), направленных на решение проблем, связанных с созданием прикладных систем в современном мире Интернета. Как правило, прикладной сервер состоит из некоторого набора нескольких различных технологий, таких как веб-серверы, посредники объектных запросов (Object Request Broker, ORB), программное обеспечение, ориентированное на сообщение (Message-Oriented Middleware, MOM), базы данных и пр. Сервер приложений может быть также ориентирован только на одну технологию, такую как технология распределенных объектов. Возможности серверов приложений, базирующиеся на распределенных объектах, сильно варьируются. Самыми простыми являются посредники ORB, обеспечивающие связь между клиентскими приложениями и распределенными объектами. ORB позволяют клиентским приложениям без труда находить и использовать распределенные объекты. Однако ORB часто оказываются непригодными в массивных транзакционных окружениях. Они предоставляют коммуникационный каркас для распределенных объектов, но не могут обеспечить сколько-нибудь мощную инфраструктуру, способную справиться с большим количеством пользователей и гарантировать надежную работу. Кроме этого, ORB предлагает недостаточно проработанную модель серверных компонентов, которая перекладывает обязанности по обработке транзакций, параллелизма, постоянства и другие задачи системного уровня на плечи разработчика приложения. Все эти службы не поддерживаются автоматически в ORB. Разработчики приложений должны напрямую обращаться к ним (если они доступны) или, в некоторых случаях, создавать их с нуля.

В начале 1999 года Анна Мейнс (Anne Manes)[1] ввела в обращение термин *монитор компонентных транзакций* (*Component Transaction Monitor, CTM*) для того, чтобы описать наиболее развитый прикладной сервер распределенных объектов. CTM возникли как сплав традиционных TP-мониторов и технологии ORB. Они реализуют мощную модель серверных компонентов, которая помогает разработчикам создавать, использовать и развертывать прикладные системы. CTM предлагают инфраструктуру, которая автоматически управляет транзак-

[1] Когда Мейнс ввела этот термин, она работала в Patricia Seybold Group под именем Анны Томас. Сейчас Мейнс является директором Business Strategy for Sun Microsystem – подразделения Sun Software.

циями, распределением объектов, параллелизмом, безопасностью, постоянством и контролем за ресурсами. Они способны управлять огромным числом пользователей и обеспечивать надежную работу системы и, кроме этого, представляют ценность для малых систем из-за легкости в обращении. СТМ – это законченные серверы приложений. Другие термины, применяемые к этой технологии, включают в себя: монитор объектных транзакций (Object Transaction Monitor, OTM), сервер компонентных транзакций, сервер распределенных компонентов и COM-ware. В данной книге используется термин «монитор компонентных транзакций», – из-за того, что он охватывает три ключевые характеристики этой технологии: применение компонентной модели, направленность на управление транзакциями и управление ресурсами и сервисами, которое обычно связывают с мониторами.

Определение Enterprise JavaBeans

Компания Sun Microsystems определяет Enterprise JavaBeans следующим образом:

> Архитектура Enterprise JavaBeans – это компонентная архитектура, предназначенная для разработки и развертывания основанных на компонентах распределенных бизнес-приложений. Приложения, созданные с помощью архитектуры Enterprise JavaBeans, являются масштабируемыми, ориентированными на транзакции и безопасными при работе в многопользовательском режиме. Эти приложения, однажды написанные, могут затем быть развернуты на любой серверной платформе, поддерживающей спецификацию Enterprise JavaBeans.[1]

Весьма многословно, но достаточно типично для определений, которые Sun дает многим Java-технологиям. Вы когда-нибудь читали определение для самого языка Java? Оно примерно в два раза длиннее. Здесь предлагается сокращенное определение EJB:

> Enterprise JavaBeans – это стандартная модель серверных компонентов для мониторов компонентных транзакций.

Мы подготовили почву для этого, определив кратко термины «распределенные объекты», «серверные компоненты» и «мониторы компонентных транзакций». Построим прочный фундамент для изучения Enterprise JavaBeans, для чего далее в этой главе и раскроем все эти определения.

Те, кто уже ясно представляет себе, что такое распределенные объекты, мониторы транзакций, СТМ и системы асинхронных сообщений, могут пропустить остаток этой главы и перейти к главе 2.

---

[1] Спецификация Enterprise JavaBeans версия 2.0, Copyright 2001, Sun Microsystems.

Архитектуры распределенных объектов

EJB – это компонентная модель для мониторов компонентных транзакций, основанных на технологиях распределенных объектов. Следовательно, для того чтобы понять EJB, необходимо представлять, как работают распределенные объекты. Системы распределенных объектов являются фундаментом для современных трехуровневых архитектур. Как показано на рис. 1.1, логика представления расположена на стороне клиента (первый уровень), прикладная логика размещена на промежуточном уровне (второй уровень), а другие ресурсы, такие как базы данных, – на стороне источников данных (третий уровень).

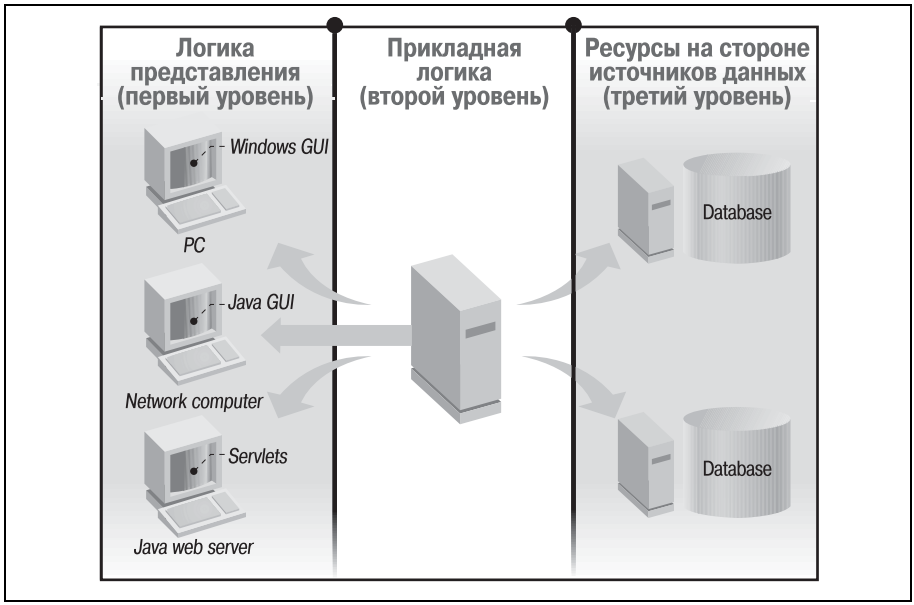

Рис. 1.1. Трехуровневая архитектура

Все протоколы распределенных объектов построены на этой же базовой архитектуре, спроектированной так, чтобы объект, расположенный на одном компьютере, казался расположенным на другом. Архитектура распределенных объектов опирается на уровень сетевых коммуникаций, который в действительности крайне прост. По существу эта архитектура состоит из трех частей: прикладного объекта, каркаса и заглушки.

Прикладной объект (*business object*) располагается на промежуточном уровне. Это экземпляр объекта, моделирующий состояние и логику некоторого реально существующего понятия, такого как человек, заказ или расчетный счет. У каждого класса прикладных объектов есть соответствующие ему классы заглушки и каркаса, специально созданные для этого типа прикладного объекта. Например, прикладной объ-

ект с именем `Person` будет иметь соответствующие классы `Person_Stub` и `Person_Skeleton`. Как показано на рис. 1.2, прикладной объект и каркас расположены на втором уровне, а заглушка – на клиентском.

Заглушка (*stub*) и *каркас* (*skeleton*) отвечают за то, чтобы прикладной объект, находящийся на промежуточном уровне, выглядел так, как будто бы он выполняется локально на клиентской машине. Это достигается за счет применения некоторого протокола, называемого *удаленным вызовом методов* (*Remote Method Invocation, RMI*). Протокол RMI служит для передачи вызовов методов через сеть. CORBA, Java RMI и Microsoft .NET пользуются своими собственными протоколами.[1] Каждый экземпляр прикладного объекта, находящийся на втором уровне, обернут (wrapped) экземпляром соответствующего ему класса каркаса. Каркас настраивается на определенный порт и IP-адрес и ожидает запросы от заглушки, находящейся на клиентской машине и соединяющейся с каркасом через сеть. Заглушка действует в качестве заместителя прикладного объекта на клиентской машине и отвечает за передачу запросов от клиента прикладному объекту при помощи каркаса. Рис. 1.2. иллюстрирует передачу вызова метода от клиента серверному объекту и обратно. Заглушка и каркас скрывают детали передачи, специфичные для RMI-протокола, от клиента и класса реализации соответственно.

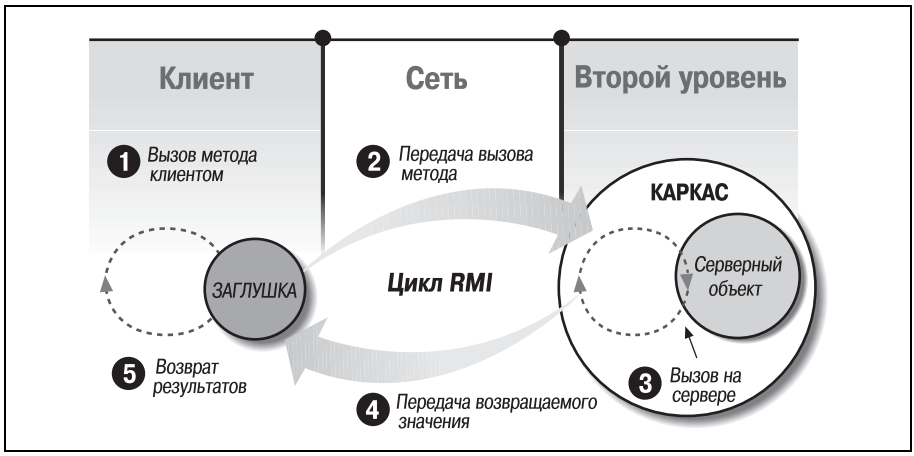

Рис. 1.2. Цикл RMI

Прикладной объект реализует открытый интерфейс, объявляющий его прикладные методы. Заглушка реализует тот же самый интерфейс, что и прикладной объект, но методы заглушки не содержат

[1] Аббревиатура RMI не относится только к Java RMI. В этом разделе термин RMI используется для описания протоколов распределенных объектов вообще. Java RMI представляет собой версию протокола распределенных объектов языка программирования Java.

прикладной логики. Вместо этого прикладные методы заглушки реализуют некоторые сетевые операции, требуемые для направления запроса прикладному объекту и принятия от него результата. Когда клиент вызывает прикладной метод заглушки, запрос передается через сеть посредством отправки каркасу имени вызванного метода и значений входных параметров. Когда каркас принимает входной поток, он анализирует его и определяет запрашиваемый метод, а затем вызывает соответствующий метод прикладного объекта. Все значения, возвращаемые вызванным методом прикладного объекта, передаются обратно заглушке через каркас. Заглушка затем возвращает эти значения клиентскому приложению, как будто они были обработаны локальной прикладной логикой.

Обкатка своего собственного объекта

Наилучший способ проиллюстрировать работу объекта состоит в том, чтобы показать, как вы можете сами реализовать распределенный объект с помощью собственного протокола распределенных объектов. Это дает вам некоторое представление о том, как работает настоящий протокол распределенных объектов, например CORBA. Однако существующие системы распределенных объектов, такие как DCOM, CORBA и Java RMI, гораздо сложнее и мощнее, чем этот простой пример, который мы сейчас разработаем. Система распределенных объектов, которую мы создадим в этой главе, является исключительно иллюстративной. Она не представляет собой ни настоящую технологию, ни часть Enterprise JavaBeans. Ее цель – дать читателю некоторое понимание механизма работы более изощренных систем распределенных объектов.

Здесь приведен очень простой прикладной объект с именем `PersonServer`, который реализует интерфейс `Person`. Интерфейс `Person` охватывает концепцию прикладного объекта, соответствующего определенному человеку. У него есть два метода: `getAge()` и `getName()`. В реальном приложении мы, возможно, определили бы более сложное поведение для объекта Person, но для этого примера вполне достаточно двух методов:

```
public interface Person {
    public int getAge() throws Throwable;
    public String getName() throws Throwable;
}
```

Реализация этого интерфейса `PersonServer` не содержит ничего неожиданного. Она определяет прикладную логику и состояние объекта Person:

```
public class PersonServer implements Person {
    int age;
    String name;
```

```
    public PersonServer(String name, int age){
        this.age = age;
        this.name = name;
    }
    public int getAge(){
        return age;
    }
    public String getName(){
        return name;
    }
}
```

Сейчас нам нужен некоторый способ сделать объект PersonServer доступным для удаленного клиента. Это является обязанностью объектов Person_Skeleton и Person_Stub. Интерфейс Person описывает независящее от реализации понятие человека. Оба объекта – и PersonServer, и Person_Stub – реализуют интерфейс Person, поскольку они оба предназначены для поддержки понятия «человек». Объект PersonServer реализует интерфейс для того, чтобы обеспечить существующую прикладную логику и состояние. Person_Stub реализует данный интерфейс для того, чтобы походить на прикладной объект Person на стороне клиента и передавать запросы обратно каркасу, который, в свою очередь, должен передать их самому объекту. Здесь показано, как выглядит эта заглушка.

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;

public class Person_Stub implements Person {
    Socket socket;

    public Person_Stub() throws Throwable {
        /* Устанавливаем сетевое соединение с каркасом.
            Используем имя "localhost" или IP-адрес каркаса,
            если он находится на другой машине */
        socket = new Socket("localhost",9000);
    }
    public int getAge() throws Throwable {
        // Во время вызова этого метода имя метода передается каркасу
        ObjectOutputStream outStream =
            new ObjectOutputStream(socket.getOutputStream());
        outStream.writeObject("age");
        outStream.flush();
        ObjectInputStream inStream =
            new ObjectInputStream(socket.getInputStream());
        return inStream.readInt();
    }
    public String getName() throws Throwable {
        // Во время вызова этого метода имя метода передается каркасу
```

```
            ObjectOutputStream outStream =
                new ObjectOutputStream(socket.getOutputStream());
            outStream.writeObject("name");
            outStream.flush();
            ObjectInputStream inStream =
                new ObjectInputStream(socket.getInputStream());
            return (String)inStream.readObject();
        }
    }
```

Когда вызывается метод объекта `Person_Stub`, создается и направляется каркасу маркер (token), представляющий собой объект `String`. В этой посылке указывается вызываемый метод заглушки. Каркас анализирует посылку, определяет метод, вызывает соответствующий метод прикладного объекта и отправляет результат обратно. Когда заглушка считывает ответ от каркаса, она анализирует его значение и возвращает клиенту. С точки зрения клиента, заглушка обрабатывает запрос локально. Посмотрим на каркас:

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;
import java.net.ServerSocket;

public class Person_Skeleton extends Thread {
    PersonServer myServer;

    public Person_Skeleton(PersonServer server){
        // Получаем ссылку на прикладной объект, представленный каркасом
        this.myServer = server;
    }
    public void run(){
        try {
            // Создаем серверный сокет на порту 9000.
            ServerSocket serverSocket = new ServerSocket(9000);
            // Ждем и получаем сетевое соединение с заглушкой
            Socket socket = serverSocket.accept();
            while (socket != null){
                // Создаем входной поток для приема запросов от заглушки
                ObjectInputStream inStream =
                    new ObjectInputStream(socket.getInputStream());
                // Читаем очередной запрос метода от заглушки
                // Блокируем поток на время приема
                String method = (String)inStream.readObject();
                // Определяем тип запрошенного метода
                if (method.equals("age")){
                    // Вызываем прикладной метод серверного объекта
                    int age = myServer.getAge();
                    // Создаем выходной поток для отправки результата
                    // обратно заглушке
                    ObjectOutputStream outStream =
```

```
                    new ObjectOutputStream(socket.getOutputStream());
                // Отправляем результат заглушке
                outStream.writeInt(age);
                outStream.flush();
            } else if(method.equals("name")){
                // Вызываем прикладной метод серверного объекта
                String name = myServer.getName();
                // Создаем выходной поток для отправки результата
                // обратно заглушке
                ObjectOutputStream outStream =
                    new ObjectOutputStream(socket.getOutputStream());
                // Отправляем результат заглушке
                outStream.writeObject(name);
                outStream.flush();
            }
        }
    } catch(Throwable t) {t.printStackTrace();System.exit(0); }
  }
  public static void main(String args [] ){
    // Получаем уникальный экземпляр Person
    PersonServer person = new PersonServer("Richard", 36);
    Person_Skeleton skel = new Person_Skeleton(person);
    skel.start();
  }
}
```

Объект `Person_Skeleton` перенаправляет запросы, получаемые от заглушки, прикладному объекту `PersonServer`. Естественно, что `Person_Skeleton` тратит все свое время на ожидание запроса от заглушки. Полученный запрос анализируется и направляется соответствующему методу объекта `PersonServer`. Значение, возвращаемое прикладным объектом, направляется затем обратно заглушке, которая возвращает его клиенту, как будто оно было получено локально.

Итак, создав весь механизм, давайте посмотрим на простую клиентскую программу, использующую объект `Person`:

```
public class PersonClient {
  public static void main(String [] args){
    try {
      Person person = new Person_Stub();
      int age = person.getAge();
      String name = person.getName();
      System.out.println(name+" is "+age+" years old");
    } catch(Throwable t) {t.printStackTrace();}
  }
}
```

Это клиентское приложение показывает, как заглушка используется клиентом. Если не считать создания экземпляра `Person_Stub` в самом

начале, клиент не знает, что прикладной объект Person в действительности – сетевой заместитель реального прикладного объекта, расположенного на промежуточном уровне. На рис. 1.3. диаграмма цикла RMI изменена, для того чтобы можно было видеть, как процесс RMI применяется к нашему коду.

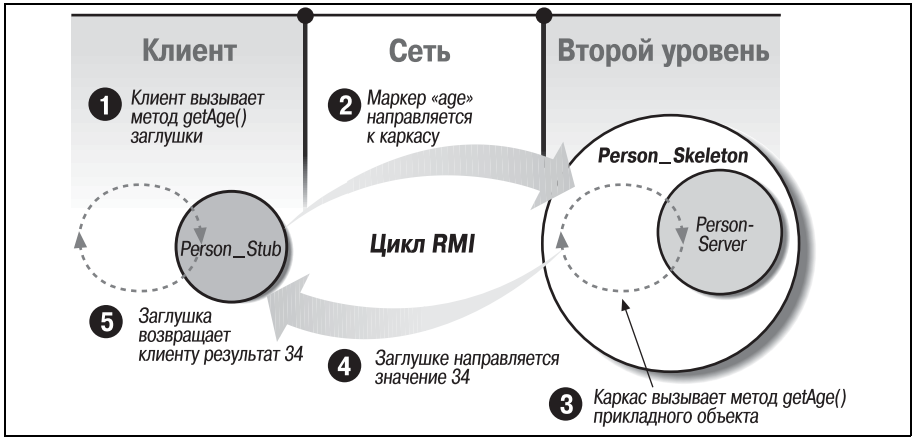

Рис. 1.3. Цикл RMI для прикладного объекта Person

Рис. 1.3. иллюстрирует реализацию цикла RMI при помощи распределенного объекта Person. RMI является основой систем распределенных объектов и отвечает за *прозрачность расположения* (*location transparency*) распределенных объектов. Прозрачность расположения означает, что действительное расположение серверного объекта – обычно в промежуточном слое – неизвестно использующему его клиенту и не имеет для него значения. В нашем примере клиент может находиться на той же машине либо на другой машине очень далеко отсюда, но взаимодействие клиента с прикладным объектом остается таким же. Прозрачность расположения является одним из самых больших преимуществ систем распределенных объектов. Хотя прозрачность и является преимуществом, во время проектирования вы не должны относиться к распределенным объектам как к локальным, из-за разницы в их производительности. В этой книге представлены удачные стратегии дизайна распределенных объектов, которые благодаря преимуществам прозрачности максимизируют производительность распределенной системы.

В данной книге заглушка на стороне клиента часто называется *удаленной ссылкой* (*remote reference*) на прикладной объект. Это позволяет нам прямо говорить о прикладном объекте и его представлении на стороне клиента.

Протоколы распределенных объектов, такие как CORBA, DCOM и Java RMI, обеспечивают гораздо более мощную инфраструктуру для распределенных объектов, чем в примере с Person. Большинство реализа-

ций протоколов распределенных объектов предоставляют средства автоматического создания соответствующих заглушек и каркасов для прикладных объектов. Это устраняет необходимость ручного создания этих конструкций и позволяет включать в заглушки и каркасы гораздо больше функциональных возможностей.

Даже при автоматической генерации заглушек и каркасов пример с `Person` лишь касается поверхности изощренного протокола распределенных объектов. Реальные протоколы, такие как Java RMI и CORBA IIOP, предоставляют обработку ошибок и исключений, передачу параметров и другие службы, например передачу контекстов транзакций и безопасности. Кроме этого, протоколы распределенных объектов поддерживают гораздо более изощренные механизмы связи между заглушками и каркасами. Прямое соединение заглушки с каркасом, используемое в примере с `Person`, – достаточно примитивно.

Реальные протоколы распределенных объектов, такие как CORBA, предоставляют также посредников объектных запросов (Object Request Broker, ORB), позволяющих клиентам находить и соединяться с распределенными объектами через сеть. ORB – это каркас (или коммутатор) для распределенных объектов. Кроме управления соединениями, ORB обычно используют систему имен для поиска объектов и массу других возможностей, например передачу ссылок, распределенный сбор мусора и управление ресурсами. Однако ORB ограничиваются соединениями между клиентами и распределенными прикладными объектами. Хотя они и могут поддерживать службы, такие как управление транзакциями и безопасность, службы эти не используются автоматически. При работе с ORB большая часть обязанностей по обеспечению функциональных возможностей системного уровня и взаимодействию различных служб ложится на плечи разработчика приложения.

Модели компонентов

Термин «модели компонентов» имеет много различных толкований. Enterprise JavaBeans определяет модель *серверных* (*server-side*) *компонентов*. Посредством набора классов и интерфейсов из пакета `java.ejb` разработчики могут создавать, собирать и развертывать компоненты, соответствующие спецификации EJB.

Исходная (первоначальная) технология JavaBeans также является компонентной моделью, но не моделью серверных компонентов, как EJB. Фактически, кроме общего имени «JavaBeans», эти две компонентные модели абсолютно ничем не связаны. В прошлом во многих публикациях EJB называли расширением исходной JavaBeans, но это неверное представление. Эти два программных интерфейса служат совершенно разным целям, и EJB не расширяет и не использует исходную компонентную модель JavaBeans.

Технология JavaBeans предназначена для *внутри*процессного (*intra*process) использования, тогда как EJB разрабатывалась для *меж*процессных (*inter*process) компонентов.

JavaBeans может быть использована для решения различных проблем, но в основном она применяется для построения клиентских приложений путем соединения визуальных (GUI) и невизуальных компонентов. Это превосходная компонентная модель, возможно, лучшая из всех когда-либо созданных для внутрипроцессной разработки, но она не является моделью серверных компонентов. EJB, с другой стороны, специально разрабатывалась для решения проблем, связанных с управлением распределенными прикладными объектами в трехуровневой архитектуре.

Но если технологии JavaBeans и Enterprise JavaBeans – совершенно разные, почему тогда они обе были названы компонентными моделями? В этом контексте компонентная модель определяет набор правил, определяющих взаимодействие между разработчиком компонентов и системой, содержащей эти компоненты. Эти правила определяют, как следует разрабатывать и упаковывать компоненты. Когда компонент определен, он становится независимым программным модулем, который может распространяться отдельно и использоваться в других приложениях. Компоненты разрабатываются для определенных целей, а не для определенного приложения. В исходной технологии JavaBeans компоненты могли быть кнопками или электронными таблицами (spreadsheet) и использоваться в любом графическом приложении, в соответствии с правилами, заданными компонентной моделью JavaBeans. В модели EJB компонент может быть прикладным объектом, представляющим клиент, развертываемым на любом сервере EJB и используемым для создания любого приложения, нуждающегося в таком объекте. К другим типам компонентных моделей Java относятся сервлеты, страницы JSP и апплеты.

Мониторы компонентных транзакций (CTM)

Технология CTM выросла из технологий ORB и TP-мониторов. CTM – это гибрид двух технологий, представляющий собой мощную платформу для распределенных объектов. Для того чтобы лучше понять, что такое CTM, рассмотрим сильные и слабые стороны TP-мониторов и ORB.

Мониторы обработки транзакций

Мониторы обработки транзакций (Transaction Processing, TP) появились около 30 лет назад (Customer Information Control System, CICS, была представлена в 1968 году) и сделались мощными, быстрыми серверными платформами для надежных приложений. Некоторые TP-продукты, такие как CICS и TUXEDO, должны быть вам знакомы.

Мониторы TP являются операционными системами для приложений, написанных на языках типа COBOL. «Операционная система» может показаться странным названием для монитора TP, но оно вполне оправданно, поскольку монитор управляет всей средой приложения. Системы с мониторами TP автоматически управляют всем окружением, в котором выполняется приложение, включая транзакции, управление ресурсами и восстановление после сбоев. Приложения, выполняющиеся внутри мониторов TP, пишутся на процедурных языках (т. е. COBOL и C) и обычно доступны через *систему сетевых сообщений* (*network messaging*) и *удаленные вызовы процедур* (*remote procedure calls, RPC*). Система сообщений позволяет клиенту посылать сообщения монитору TP, требуя запустить некоторое приложение с заданными параметрами. Это похоже на принцип работы модели событий Java. Сообщения могут быть синхронными или асинхронными. Это означает, что источник сообщений не должен ждать ответа. RPC, которая является предком RMI, – это механизм распределения, позволяющий клиентам вызывать процедуры приложения, работающего внутри монитора TP, как если бы процедура выполнялась локально. Основное различие между RPC и RMI состоит в том, что RPC используется в *процедурных* приложениях, а RMI – в системах распределенных *объектов*. С помощью RMI могут вызываться методы заданных экземпляров объектов, т. е. заданных прикладных сущностей. С помощью RPC клиент может вызвать процедуру определенного типа приложения, где нет понятия экземпляра объекта. Таким образом, RMI представляет собой объектно-ориентированный механизм, а RPC – процедурный.

Мониторы TP существуют уже достаточно долгое время, поэтому технология, лежащая в их основе, стала прочной, как камень. Вот почему они сегодня применяются во многих прикладных системах. Но мониторы TP не являются объектно-ориентированными. Они работают с процедурным кодом, способным выполнять сложные задачи, но не имеющим представления об объектах. Доступ к мониторам TP через RPC похож на вызов статических методов, здесь не существует понятия уникального объекта. Кроме этого, по той причине, что мониторы TP основаны на процедурных приложениях, а не на объектах, прикладная логика в мониторах TP не является гибкой, расширяемой или многократно используемой, как у прикладных объектов в системах с распределенными объектами.

Посредники объектных запросов

Благодаря системам распределенных объектов доступ к уникальным объектам, имеющим состояние и идентификатор, возможен из всех точек сети. Технологии распределенных объектов, такие как CORBA и Java RMI, отличаются от RPC одной существенной чертой: мы вызываем метод, принадлежащий экземпляру компонента, а не прикладной

ную процедуру. Распределенные объекты разворачиваются обычно на каком-либо ORB, отвечающем за предоставление клиентским приложениям возможности легкого поиска этих объектов.

Однако ORB не являются «операционными системами» для распределенных объектов. Они представляют собой просто коммуникационный каркас, используемый для доступа и взаимодействия с уникальными удаленными объектами. При разработке приложения на основе распределенных объектов, использующего ORB, вся ответственность за реализацию параллелизма, транзакций, механизмов управления ресурсами и защиты от сбоев ложится на плечи программиста. Эти службы могут поддерживаться ORB, но разработчик приложения отвечает за объединение их с прикладными объектами. В ORB отсутствует понятие операционной системы, в которой задачи системного уровня обрабатываются автоматически. Недостаток неявной инфраструктуры системного уровня создает многочисленные проблемы для разработчика приложения. Создание такой инфраструктуры, требующей обработки параллелизма, транзакций, безопасности, постоянства и кроме этого нуждающейся в поддержке большого количества пользователей, является задачей, достойной Геркулеса, которую способны решить очень немногие корпоративные коллективы разработчиков.

CTM: гибрид ORB и TP-мониторов

Когда преимущества распределенных объектов стали очевидными, количество внедряемых систем, применяющих ORB, стало быстро увеличиваться. ORB поддерживают распределенные объекты, используя некую «сырую» модель серверных компонентов, позволяющую распределенным объектам подключаться к коммуникационному каркасу, но в то же время они не обеспечивают полную поддержку транзакций, безопасности, постоянства и управления ресурсами. Эти службы должны быть явно доступны распределенным объектам через программные интерфейсы, что приводит к большей сложности и зачастую к более серьезным проблемам при развертывании. Кроме этого, стратегии управления ресурсами, такие как подкачка экземпляров, пул ресурсов и активация не поддерживаются вовсе. Такие виды стратегии делают возможным масштабирование систем распределенных объектов, улучшают производительность и пропускную способность и являются более понятными. Без автоматической поддержки управления ресурсами разработчик приложения вынужден применять «доморощенные» решения по управлению ресурсами, требующие отличного знания систем распределенных объектов. ORB проигрывают в отношении сложности управления компонентами в крупномасштабных прикладных средах, в тех областях, где мониторы TP всегда считались непревзойденными.

После тридцатилетнего опыта работы с мониторами TP компании, такие как IBM и BEA, начали разработку гибрида систем ORB и TP-мо-

ниторов, названного мониторами компонентных транзакций (CTM). Этот тип серверов приложений соединяет в себе гибкость и доступность систем распределенных объектов, основанных на ORB с мощностью «операционных систем» мониторов TP. CTM предоставляют комплексную среду для серверных компонентов, обеспечивая автоматическое управление параллелизмом, транзакциями, распределением объектов, выравниванием загрузки, безопасностью и ресурсами. Хотя разработчики приложений по-прежнему должны быть знакомы с этими возможностями, в случае применения CTM им нет необходимости явно реализовывать их.

Основными особенностями CTM являются распределенные объекты, инфраструктура, включающая управление транзакциями и другие службы, и модель серверных компонентов. CTM поддерживают эти особенности в разной степени. При выборе CTM не так важно найти самый мощный и многофункциональный, как тот, который лучшим образом удовлетворяет вашим требованиям. Очень большие и мощные CTM могут стоить чрезвычайно дорого, а их применение для небольших проектов избыточно. CTM используется в нескольких областях, включая реляционные базы данных, серверы приложений, веб-серверы, ORB для CORBA и мониторы TP. Каждый производитель предлагает продукты, отражающие его область компетенции. Однако для начинающего выбор CTM с поддержкой компонентной модели Enterprise JavaBeans может быть важнее любого другого набора функциональных возможностей. По той причине, что Enterprise JavaBeans является независимой от реализации, выбор EJB CTM даст прикладной системе в случае необходимости возможность масштабирования на более крупный CTM. Мы обсудим важность EJB как стандартной компонентной модели для CTM далее в этой главе.

Сравнение с реляционными базами данных

В этой главе обсуждению CTM уделяется так много времени потому, что они важны для определения EJB. Обсуждение CTM еще не закончено, но для того чтобы дальнейшее изложение стало понятным, мы сравним CTM с реляционной базой данных.

Реляционные базы данных предоставляют разработчикам приложений простую среду разработки в сочетании с мощной инфраструктурой данных. Разработчик приложения, использующего базу данных, может задавать расположение таблиц, решать, какие столбцы следует сделать первичными ключами, определить индексы и хранимые процедуры, но он не должен разрабатывать алгоритмы индексации, анализатор SQL и системы управления курсорами. Реализация этих типов функциональности системного уровня возложена на производителей баз данных, а разработчик просто выбирает тот продукт, который наилучшим образом удовлетворяет его требованиям. Разработчиков приложений интересует только организация их прикладных данных,

а не то, как работает процессор базы данных. Для разработчика приложений написание реляционной базы данных с нуля стало бы напрасной тратой времени, ведь Microsoft и Oracle уже предоставляют их.

Для эффективной работы распределенные объекты требуют такого же управления системным уровнем со стороны СТМ, как и прикладные данные от реляционных баз данных. Функциональные возможности системного уровня, а именно параллелизм, управление транзакциями и управление ресурсами, необходимы тогда, когда прикладную систему планируется применять для работы с большим количеством пользователей. Нереально и напрасно ждать от разработчика приложения, что он еще раз изобретет всю эту функциональность системного уровня, тогда как уже существуют готовые коммерческие решения.

СТМ является для прикладных объектов тем, чем реляционные базы данных являются для данных. СТМ отвечает за всю работу системного уровня, позволяя разработчикам приложений сосредоточиться на стоящих перед ними задачах. Применяя СТМ, разработчик приложения может сфокусироваться на разработке и реализации прикладных объектов без необходимости тратить тысячи часов на разработку инфраструктуры, в которой эти объекты будут выполняться.

СТМ и модели серверных компонентов

СТМ требуют, чтобы прикладные объекты придерживались модели серверных компонентов, реализованной производителем. Удачная модель компонентов представляет собой решающий фактор успеха разрабатываемого проекта, т. к. она определяет, насколько легко разработчик приложения может написать прикладной объект под данный СТМ. Модель компонентов является соглашением, определяющим обязанности СТМ и прикладных объектов. Имея дело с удачной компонентной моделью, разработчик знает, что можно ожидать от СТМ, а СТМ «понимает», как следует управлять прикладными объектами. Модели серверных компонентов хорошо подходят для описания обязанностей разработчика приложения и производителя СТМ.

Модели серверных компонентов опираются на определенную спецификацию. Если компонент придерживается этой спецификации, он может быть использован СТМ. Отношения между серверным компонентом и СТМ похожи на отношения между CD-диском и CD-проигрывателем. Если компонент (CD-диск) придерживается спецификации проигрывателя, он может быть воспроизведен на нем.

Отношения СТМ с его компонентной моделью похожи также на отношения между железнодорожной системой и поездом. Железнодорожная система управляет окружением поезда, предоставляя дополнительные пути с целью равномерного распределения нагрузки, множественные пути для обеспечения параллельной работы и систему управления трафиком для управления ресурсами. Железная дорога

предоставляет инфраструктуру, на которой работает поезд. Точно так же CTM предоставляет серверным компонентам всю инфраструктуру, необходимую для поддержки параллелизма, транзакций, распределения загрузки и т. д.

Поезда на железной дороге похожи на серверные компоненты: все они выполняют различные задачи, но выполняют их, используя один базовый проект. Поезда ориентированы на выполнение задач, таких как перевозка автомобилей, а не на управление их окружением. Для машиниста – лица, управляющего поездом, интерфейс управления достаточно прост: тормоза и дроссель. Для разработчика приложения интерфейсы серверного компонента ограничены похожим образом.

Различные CTM могут реализовывать различные компонентные модели точно так же, как различные железные дороги могут иметь различные типы поездов. А различия между моделями компонентов варьируются так же, как железнодорожные системы имеют пути разной ширины и различные системы управления, но базовые операции CTM остаются одними и теми же. Все они гарантируют, что управление прикладными объектами обеспечит поддержку большого числа пользователей в критичных ситуациях. Это означает, что ресурсы, параллельность, транзакции, безопасность, постоянство, баланс загрузки и распределение объектов могут быть обработаны автоматически, ограничив разработчика приложения простым интерфейсом. Это позволяет разработчику приложения концентрироваться на прикладной логике, а не на корпоративной инфраструктуре.

Среда .NET от Microsoft

Корпорация Microsoft была первым производителем, начавшим распространять CTM. Первоначально названный сервером транзакций (Microsoft Transaction Server, MTS), он позднее был переименован в COM+. Microsoft COM+ основан на модели компонентных объектов (Component Object Model, COM), первоначально разработанной для использования в настольных компьютерах, но со временем переродившейся в сервис в виде модели серверных компонентов. Для распределенного доступа клиента COM+ используют распределенную модель компонентных объектов (Distributed Component Object Model, DCOM).

В 1996 году, когда MTS был представлен впервые, он казался весьма впечатляющим, поскольку предоставлял универсальное окружение для прикладных объектов. При помощи MTS разработчики приложений могли писать компоненты COM, не заботясь о вопросах системного уровня. Если прикладной объект был разработан в соответствии с моделью COM, то MTS (а сейчас COM+) заботился обо всем остальном, включая управление транзакциями, параллелизм и управление ресурсами.

Недавно COM+ стал частью новой среды – .NET (.NET Framework) от Microsoft. Базовая функциональность, предоставляемая сервисами COM+, фактически сохранилась в .NET, но перед разработчиком она теперь предстает в существенно изменившейся «ипостаси». Разработчик вместо того, чтобы писать компоненты в виде объектов COM, в среде .NET строит приложения в виде *управляемых объектов* (*managed object*). Все управляемые объекты и фактически весь код, написанный под .NET, зависит от общеязыковой среды времени выполнения (Common Language Runtime, CLR). Для разработчиков на Java среда CLR во многом напоминает виртуальную машину (VM) Java, а управляемые объекты представляют собой аналоги экземпляров классов Java, т. е. объектов Java.

Хотя среда .NET и предоставляет много интересных особенностей, в качестве открытого стандарта они использоваться не могут. Службы COM+ в среде .NET являются собственным СТМ фирмы Microsoft. Это может быть не так уж плохо, поскольку .NET обещает работать хорошо, а платформа Microsoft широко распространена. Кроме того, поддержка средой .NET простого протокола доступа к объектам (Simple Object Access Protocol, SOAP) позволяет прикладным объектам из среды .NET связываться с объектами любой другой платформы, написанными на любом языке. Это потенциально может сделать прикладные объекты из среды .NET универсально доступными – возможность, от которой не так легко отказаться.

Однако если какая-то компания намеревается развертывать серверные компоненты не на платформе Microsoft, .NET не будет жизнеспособным решением. Кроме того, службы COM+ в среде .NET ориентированы на компоненты без состояния, в них нет встроенной поддержки для постоянных транзакционных объектов (persistent transaction objects). Хотя объекты без состояния могут предложить более высокую производительность, прикладным системам требуется такой тип гибкости, предлагаемый СТМ, который включает компоненты с состоянием и постоянные компоненты.

EJB и CORBA СТМ

До осени 1997 года мониторы СТМ, можно сказать никто, кроме Microsoft, не выпускал. Многообещающие продукты от IBM, BEA и Hitachi были еще «на чертежной доске», а MTS уже присутствовал на рынке. Все остальные проекты, оставаясь лишь проектами, имели одну общую черту: в качестве службы распределенных объектов они использовали технологию CORBA.

Большинство СТМ «не от Microsoft» были направлены на то, чтобы стать своевременными и более открытыми стандартами CORBA, благодаря этому они должны были работать на несовместимых с Microsoft платформах и поддерживать несовместимые с Microsoft клиенты. Благодаря тому что CORBA не зависит как от языка, так и от платформы,

производители СТМ могли дать своим клиентам больше альтернативных вариантов при реализации.[1] Проблемой разработок CORBA СТМ было то, что они поддерживали различные модели серверных компонентов. Другими словами, если компонент разрабатывался под СТМ одного производителя, то уже нельзя было повернуть назад и использовать этот же самый компонент на СТМ от другого производителя. Компонентные модели были слишком разные.

Для конкуренции с MTS от Microsoft, удерживавшим лидерство вплоть до 1997 года (к тому времени он уже существовал около года), СТМ, ориентированные на CORBA, нуждались в конкурентоспособной идее. Одной из проблем, с которой столкнулись СТМ, была раздробленность рынка CORBA, в котором продукт каждого производителя отличался от своего соседа. Раздробленный рынок не был нужен никому, поэтому производителям CORBA СТМ был необходим стандарт, вокруг которого можно было бы сплотиться. Кроме протокола CORBA, наиболее очевидным требуемым стандартом была компонентная модель, которая позволила бы клиентам и третьим производителям разрабатывать собственные прикладные объекты для одной спецификации, способные работать на любой CORBA СТМ. Microsoft, разумеется, проталкивала свою компонентную модель в качестве стандарта, который был привлекателен, поскольку MTS представлял собой реально работающий продукт, но он не поддерживал CORBA. Object Management Group (OMG) – те же самые люди, которые создали стандарт CORBA, определили альтернативную модель серверных компонентов. Модель была многообещающей, поскольку наверняка прекрасно ужилась бы с CORBA, но OMG разрабатывала стандарт медленно, слишком медленно для развивающегося рынка СТМ.[2]

В 1997 году компания Sun Microsystems разработала самый многообещающий стандарт для серверных компонентов: Enterprise JavaBeans. Sun предложила несколько важных преимуществ. Во-первых, Sun была уважаемой компанией, получившей известность по работе с производителями с целью определения ориентированных на Java программных интерфейсов для общераспространенных сервисов. Во-вторых,

---

[1] Недавнее появление SOAP поставило под вопрос будущее протокола CORBA IIOP (Internet Inter-Operability Protocol). Очевидно, что эти два протокола борются между собой за право стать стандартным, независимым от языка протоколом для распределенных вычислений. IIOP существует уже около семи лет и, следовательно, более проработан, но SOAP может быстро его нагнать, используя опыт, накопленный при его создании.

[2] Компонентная модель CORBA СТМ, CORBA Component Model (CCM), в конце концов была завершена. В целом она получила довольно слабое признание и была вынуждена включить Enterprise JavaBeans как часть своей компонентной модели просто для того, чтобы быть нужной и привлекательной.

Sun имела привычку перенимать лучшие идеи и затем выпускать реализацию Java открытым стандартом – и, как правило, удачно. Программный интерфейс взаимодействия с базами данных, названный JDBC, тому прекрасный пример. В-третьих, опираясь в основном на ODBC от Microsoft, JDBC предложил производителям более гибкую модель для включения в нее их собственных драйверов доступа к базам данных. Ну и, кроме того, разработчики нашли интерфейс JDBC более легким в работе, чем ODBC. Sun проделала эти же вещи и со своими более новыми технологиями, такими как программный интерфейс JavaMail и Java Naming and Directory Interface (JNDI). Эти технологии все еще находились в разработке, но сотрудничество между производителями обнадеживало, а открытость программных интерфейсов была очень привлекательна.

Хотя CORBA предложила открытый стандарт, она пыталась стандартизировать низкоуровневые механизмы, а именно безопасность и транзакции. Производители не могли позволить себе переписать заново существующие продукты вроде TUXEDO или CICS в соответствии со стандартами CORBA. EJB обошла эту проблему, объявив, что способ реализации низкоуровневых служб не имеет значения. Имеет значение только то, что все эти механизмы, применяемые к компонентам, соответствуют спецификации – гораздо более удобное решение для существующих и будущих производителей СТМ. Кроме того, язык Java предлагает несколько довольно соблазнительных преимуществ, не все из которых чисто технические. Во-первых, Java была очень популярной технологией, и, просто сделав свой продукт совместимым с Java, можно было обеспечить ему прорыв на рынок. Java является более или менее независимой от платформы, а компонентные модели, определенные в языке Java, обладают четко выраженными коммерческими и техническими преимуществами.

Анонсировав Enterprise JavaBeans, корпорация Sun не стала почивать на лаврах. Ее инженеры продолжают работать с несколькими ведущими производителями над созданием гибкого и открытого стандарта, под который производители могли бы легко адаптировать свои существующие продукты. Это было трудной задачей из-за того, что у производителей имелись различные виды серверов, включающих веб-серверы, серверы реляционных баз данных, серверы приложений и ранние версии СТМ. Похоже на то, что никто не хотел жертвовать своей архитектурой ради общего блага, но постепенно производители согласились с моделью, которая была достаточно гибкой для согласования различных реализаций и в то же время достаточно надежной для поддержки реальных прикладных разработок. В декабре 1997 году Sun Microsystems выпустила первую рабочую спецификацию Enterprise JavaBeans – EJB 1.0, и производители сплотились вокруг этой модели серверных компонентов как никогда раньше.

Преимущества стандартной модели серверных компонентов

Итак, что же дает стандартная модель серверных компонентов? Все достаточно просто. Это означает, что можно разработать прикладные объекты, используя прикладную модель Enterprise JavaBeans, и рассчитывать на то, что они будут работать на любом СТМ, с полной поддержкой спецификации EJB. Это достаточно смелое заявление, т. к. оно практически устраняет самую большую проблему, с которой столкнулись бы потенциальные потребители СТМ-продуктов, основанных на CORBA: зависимость от определенного производителя. Работая со стандартной моделью серверных компонентов, клиенты могут решить использовать один СТМ, совместимый с EJB, т. к. будут знать, что смогут прейти на более хороший СТМ, если он станет доступным. Очевидно, что осторожность должна быть проявлена в случае применения собственного расширения, разработанного производителем, но это не является новостью. Даже в индустрии баз данных, которая использует стандарт SQL уже пару десятилетий, существует масса дополнительных собственных расширений.

Существование стандартной модели серверных компонентов кроме независимости от реализации имеет еще несколько преимуществ. Стандартная компонентная модель дает возможность развития «третьих» продуктов. Если EJB поддерживает большое количество производителей, для производителей программных продуктов создание добавочных (add-on) продуктов и библиотек компонентов становится более привлекательным. Индустрия информационных технологий уже была свидетелем такого типа индустрий, выросших вокруг других стандартов, таких как SQL. Сотни добавочных продуктов можно в настоящий момент приобрести для расширения прикладных систем, хранящих свои данные в SQL-совместимых реляционных базах данных. Типичными примерами являются средства для создания отчетов и хранилища данных. Индустрия графических компонентов также видела рост своих третьих продуктов. Для моделей графических компонентов, таких как ActiveX от Microsoft и JavaBeans от Sun, существует мощный рынок библиотек компонентов.

Для Enterprise JavaBeans сегодня существует множество третьих продуктов. Добавочные продукты для обработки кредитных карт, доступа к существующим базам данных и другие прикладные сервисы были представлены для различных EJB-совместимых систем. Эти продукты делают разработку систем EJB проще и быстрее, чем другие альтернативы, сообщая компонентной модели EJB одинаковую привлекательность для корпоративных IS и для производителей серверов. Рынок компонентов EJB растет в нескольких областях, включая торговлю, финансы, образование, создание веб-страниц и т. д.

EJB 2.0: Асинхронные сообщения

Кроме поддержки распределенных прикладных объектов на базе RMI Enterprise JavaBeans поддерживает современную технологию асинхронных сообщений. Асинхронные сообщения – это важная парадигма распределенных вычислений, ставшая одной из главных частей EJB.

Система асинхронных сообщений позволяет двум или более приложениям обмениваться информацией в форме сообщений. Сообщение в этом случае – самодостаточный пакет прикладных данных и заголовка сетевого маршрута. Прикладные данные, содержащиеся в сообщении, могут быть какими угодно, в зависимости от прикладного сценария, и обычно содержат информацию о некоторой прикладной транзакции. В корпоративных системах передачи сообщений сообщения информируют приложения о возникновении некоторого события или происшествия в другой системе.

Сообщения передаются от одного приложения к другому по сети, с помощью ориентированного на сообщения программного обеспечения промежуточного уровня (Message-Oriented Middleware, MOM). Продукты MOM гарантируют, что сообщения должным образом будут доставлены приложениям. Кроме этого, MOM обычно обеспечивают отказоустойчивость, выравнивание нагрузки, масштабируемость и поддержку транзакций для предприятий, которым необходим надежный обмен большим количеством сообщений.

Производители MOM используют для обмена сообщениями различные форматы сообщений и сетевые протоколы, но основные принципы одинаковы. С помощью программного интерфейса сообщение, снабженное прикладными данными и информацией о маршруте, создается, а затем отсылается. Тот же самый интерфейс используется и для приема сообщения, порожденного другим приложением.

Во всех современных корпоративных системах передачи сообщений приложения обмениваются сообщениями через виртуальные каналы, называемые *пунктами назначения* (*destinations*). Когда мы посылаем сообщение, оно направляется в пункт назначения, а не указанному приложению. Любое приложение, которое подписалось или зарегистрировалось в этом пункте назначения, может получить данное сообщение. В результате применения такого подхода приложения, отсылающие сообщения, получаются развязанными (decoupled). Отправители и получатели не связаны друг с другом никоим образом и могут отсылать и принимать сообщения как им удобно.

В Enterprise JavaBeans 2.0 функциональные возможности MOM встроены в компонентную модель СТМ. Эта интеграция расширит «операционную систему» платформы EJB так, что она может поддерживать и RMI, и асинхронные сообщения. EJB 2.0 поддерживает асинхронные сообщения, используя службу сообщений Java (Java Message Service)

и новый тип компонента, называемый компонентом, управляемым сообщениями.

Служба сообщений Java

Каждый производитель MOM реализует свой собственный сетевой протокол, маршрутизацию и административные средства, но основные принципы программного интерфейса разработчика, представляемого различными MOM, остаются одинаковыми. Эта сходность программных интерфейсов делает возможным существование службы сообщений Java (JMS).

JMS – это ориентированный на производителя программный интерфейс Java, который может использоваться многими различными производителями MOM. JMS очень похожа на JDBC тем, что разработчики приложений могут многократно использовать одинаковый программный интерфейс для доступа к множеству различных систем. Если производитель предоставит для JMS совместимый с ним драйвер, то средствами программного интерфейса JMS можно организовать посылку и прием сообщений от этого производителя. Например, один и тот же программный интерфейс позволяет организовать посылку сообщений с помощью систем SonicMQ от Progress и MQSeries от IBM.

Компоненты, управляемые сообщениями

Все производители JMS представляют разработчикам приложений одинаковые программные интерфейсы для посылки и приема сообщений, а иногда они предлагают и некоторую компонентную модель для создания маршрутизаторов, способных принимать и передавать сообщения. Однако такие компонентные модели уникальны для каждого производителя MOM и не обладают переносимостью.

В Enterprise JavaBeans 2.0 появился новый тип компонентов, называемых *компонентами, управляемыми сообщениями* (*message-driven bean*), и представляющий собой разновидность стандартного компонента JMS. Он может принимать и посылать асинхронные сообщения JMS, а поскольку он совместим с другими типами компонентов RMI (т. е. объектными и сеансовыми компонентами), он также может взаимодействовать с компонентами RMI.

В EJB 2.0 компоненты, управляемые сообщениями, действуют в качестве точки интеграции приложений EJB, позволяющей другим приложениям посылать асинхронные сообщения, которые могут быть перехвачены и обработаны этим приложением. Это крайне важная особенность, которая позволит приложениям EJB лучше интегрироваться с уже существующими и другими частными системами.

Компоненты, управляемые сообщениями, также работают в контексте транзакций и требуют такой же инфраструктуры, как и транзакционные серверные компоненты, основанные на RMI. Как и все остальные компоненты, основанные на RMI, компоненты, управляемые сообщениями, являются важными прикладными объектами, играющими ключевую роль в маршрутизации и интерпретации запросов и координации приложения, посылающего запросы, с другими компонентами, основанными на RMI, т. е. корпоративными компонентами. Компоненты, управляемые сообщениями, хорошо подходят к CTM и представляют собой великолепное добавление к платформе EJB.

Круизы «Титан»: Вымышленный бизнес

Для того чтобы сделать дальнейшее изложение более легким и приятным, мы будем обсуждать все принципы, содержащиеся в этой книге, в контексте воображаемого бизнеса туристической фирмы с названием «Титан». Эта туристическая фирма является особенно интересным примером, т. к. включает в себя несколько разных бизнесов: в ней есть корабельные каюты, похожие на гостиничные номера, служба питания, как в ресторане, она предлагает различные развлекательные мероприятия и должна взаимодействовать с другим бюро путешествий.

Этот тип бизнеса является хорошим выбором для системы распределенных объектов из-за того, что многие пользователи этой системы географически рассредоточены. Коммерческие агенты, например, которым нужно делать заказы на рейсы кораблей «Титана», должны иметь доступ к системе заказов билетов. Поддержка множества (возможно, тысяч) туристических агентов требует мощной системы транзакций для обеспечения агентам доступа к системе заказов и гарантий того, что заказы будут сделаны правильно.

На протяжении всей книги мы будем создавать достаточно простой фрагмент системы EJB для «Титана», направленный на обработку заказов билетов на круиз. Это дает нам возможность разработать компоненты Ship (корабль), Cabin (каюта), TravelAgent (туристический агент), ProcessPayment (оплата) и другие. На протяжении этого процесса нам придется создать таблицы в реляционной базе данных для хранения данных, используемых в этом примере. Предполагается, что вы знакомы с системами управления реляционными базами данных и сможете создать таблицы в соответствии с приведенными SQL-операторами. EJB может применяться с любыми видами баз данных или старых приложений, но реляционные базы данных, по-видимому, являются наиболее широко известными из всех баз данных, поэтому мы выбрали их в качестве уровня хранения данных.

Что дальше?

Для того чтобы создавать прикладные объекты при помощи EJB, необходимо понимание жизненного цикла и архитектуры компонентов EJB. Это означает принципиальное понимание того, как EJB-компоненты становятся доступными в качестве распределенных объектов. Углубление понимания архитектуры EJB является целью следующих двух глав.

2

Обзор архитектуры

Как вы узнали в главе 1, Enterprise JavaBeans является компонентной моделью для мониторов компонентных транзакций (Component Transaction Monitors, CTM), которые представляют собой наиболее современный тип прикладных серверов, доступных сегодня. Для эффективного использования Enterprise JavaBeans необходимо понимать архитектуру EJB, для чего в книгу включены две главы на эту тему. Данная глава объясняет основу EJB: как компоненты распределяются в качестве прикладных объектов. В главе 3 объясняются службы и технологии по управлению ресурсами, поддерживаемые в EJB.

Для того чтобы обеспечить реальную гибкость, дизайн компонентов EJB должен быть хорошо продуманным. Для разработчиков приложений сборка компонентов – дело не трудное, требующее немного или совсем не требующее опыта работы со сложными проблемами системного уровня, которые часто мешают созданию трехуровневых архитектур. Кроме того что применение модели EJB облегчает процесс разработки приложений, это также предоставляет системным разработчикам (людям, которые создают серверы EJB) большую свободу в выборе пределов поддержки спецификации EJB.

Сходства различных CTM позволяют абстракции EJB быть стандартной компонентной моделью для каждого из них. CTM каждого производителя реализованы по-разному, но все они поддерживают основные сервисы и похожие технологии управления ресурсами. Эти сервисы и технологии детально описаны в главе 3, но часть инфраструктуры, поддерживающей их, рассматривается в данной главе.

Компонент Enterprise Bean

Серверные компоненты Enterprise JavaBeans бывают трех принципиально разных типов: *объектные компоненты* (*entity beans*), *сеансовые компоненты* (*session beans*) и *компоненты, управляемые сообщениями* (*message-driven beans*). И сеансовые и объектные компоненты являются серверными компонентами, основанными на RMI, к которым можно получить доступ при помощи распределенных объектных протоколов. Компоненты, управляемые сообщениями, появившиеся в EJB 2.0, – это асинхронные серверные компоненты, которые отвечают на асинхронные сообщения JMS.

Хорошее эмпирическое правило состоит в том, что объектные компоненты моделируют прикладные понятия, которые могут быть выражены существительными. Например, объектный компонент может представлять покупателя, часть оборудования, элемент инвентаря или даже место. Другими словами, объектные компоненты моделируют объекты реального мира. Эти объекты являются обычными постоянными записями в какой-либо базе данных. Нашему гипотетическому круизу будут нужны объектные компоненты, представляющие каюты, покупателей, корабли и т. п.

Сеансовые компоненты являются расширением клиентского приложения и отвечают за управление процессами и задачами. Компонент Ship предоставляет методы для непосредственного управления кораблем, но не говорит ничего о контексте, в котором эти действия выполняются. Учет пассажиров корабля требует использования компонента Ship, но он также требует много такого, что не имеет ничего общего с самим кораблем: нам необходимо иметь сведения о пассажирах, ценах на билеты, расписании и т. д. Сеансовые компоненты отвечают за взаимодействие этих элементов. Сеансовые компоненты предназначены для управления различными видами деятельности, такими как заказ билетов. Они поддерживают взаимоотношения между различными компонентами. Сеансовый компонент TravelAgent может использовать компоненты Cruise, Cabin и Customer – все это объектные компоненты – для того, чтобы сделать заказ.

Точно так же компоненты, управляемые сообщениями, в EJB 2.0 отвечают за координацию задач, включающих другие сеансовые и объектные компоненты. Главное отличие между компонентами, управляемыми сообщениями, и сеансовыми компонентами состоит в том, как к ним осуществляется доступ. В то время как сеансовый компонент предоставляет удаленный интерфейс, определяющий, какие методы могут быть вызваны, компонент, управляемый сообщениями, этого не делает. Вместо этого он подписывается или прослушивает определенные асинхронные сообщения, обрабатывает их и управляет действиями, которые выполняют другие компоненты в ответ на эти сообщения. Например, компонент ReservationProcessor, управляемый

сообщениями, мог бы принимать асинхронные сообщения, возможно, из существующей системы заказов, с помощью которых он мог бы координировать взаимодействия компонентов Cruise, Cabin и Customer для того, чтобы сделать заказ.

Действия, которые представляют сеансовые компоненты и компоненты, управляемые сообщениями, по своей сути являются кратковременными: вы начали делать заказ, вы сделали часть работы, и на этом их действие заканчивается. Сеансовые компоненты и компоненты, управляемые сообщениями, не соответствуют никаким элементам в базе данных. Очевидно, что сеансовые компоненты и компоненты, управляемые сообщениями, оказывают влияние на базу данных: в процессе выполнения заказа необходимо создать новый компонент Reservation, назначив объект Customer отдельному компоненту Cabin на отдельном компоненте Ship. Все эти изменения должны быть отражены в базе данных с помощью действий над соответствующими объектными компонентами. Сеансовые компоненты и компоненты, управляемые сообщениями, такие как TravelAgent и ReservationProcessor, отвечающие за заказ билетов на круиз, могут даже напрямую обращаться к базе данных для выполнения чтения, обновления и удаления данных. Но в базе данных отсутствуют записи TravelAgent и ReservationProcessor – если компонент выполнил заказ, он ожидает обработки другого.

Что мешает понимать разницу между отдельными типами компонентов, так это их крайняя гибкость. Главное различие компонентов в том, что объектные компоненты имеют устойчивое (persistent) состояние, а сеансовые компоненты и компоненты, управляемые сообщениями, моделируют взаимодействия и устойчивого состояния не имеют.

Классы и интерфейсы

Хороший путь к пониманию механизма создания компонентов состоит в том, чтобы изучить, как они реализуются. Для реализации объектного или сеансового компонента необходимо определить интерфейсы компонента[1], класс компонента и первичный ключ:

Удаленный интерфейс

Удаленный интерфейс определяет прикладные методы компонента, доступные из приложений, внешних по отношению к контейнеру EJB: прикладные методы, которые компонент представляет внешнему миру для того, чтобы выполнить свою работу. Он определяет соглашения и стандарты, хорошо подходящие для протоколов распределенных объектов. Удаленный интерфейс расширяет `ja-`

[1] В основном выделяются два типа компонентных интерфейсов: удаленные и локальные. Удаленные интерфейсы поддерживаются в обеих версиях EJB – 2.0 и 1.1, тогда как локальные интерфейсы были введены в версии 2.0 и не поддерживаются версией 1.1.

vax.ejb.EJBObject, который, в свою очередь, расширяет java.rmi.Remote. Он применяется сеансовыми и объектными компонентами совместно с удаленным внутренним интерфейсом.

Удаленный внутренний интерфейс

Внутренний интерфейс определяет методы жизненного цикла компонента, доступные из приложений, внешних по отношению к контейнеру EJB: методы жизненного цикла, применяемые для создания новых компонентов, удаления и поиска компонентов. Он определяет соглашения и стандарты, хорошо подходящие для протоколов распределенных объектов. Внутренний интерфейс расширяет javax.ejb.EJBHome, который, в свою очередь, расширяет java.rmi.Remote. Он используется сеансовыми и объектными компонентами совместно с удаленным интерфейсом.

EJB 2.0: локальный интерфейс

Локальный интерфейс компонента определяет прикладные методы, используемые другими компонентами, размещенными в том же контейнере EJB: прикладные методы, которые компонент предоставляет другим компонентам, находящимся в том же адресном пространстве. Он позволяет компонентам взаимодействовать без дополнительных затрат, связанных с применением протокола распределенных объектов, что улучшает их производительность. Локальный интерфейс расширяет javax.ejb.EJBLocalObject. Он используется сеансовыми компонентами и объектными компонентами совместно с локальным внутренним интерфейсом.

EJB 2.0: локальный внутренний интерфейс

Локальный внутренний интерфейс определяет методы жизненного цикла, вызываемые другими компонентами, распределенными в том же контейнере EJB, а именно методы жизненного цикла, которые компонент предоставляет другим компонентам, расположенным в том же адресном пространстве. Это позволяет компонентам взаимодействовать без дополнительных затрат, связанных с использованием протокола распределенных объектов, что улучшает их производительность. Локальный внутренний интерфейс расширяет javax.ejb.EJBLocalHome. Он используется сеансовыми и объектными компонентами совместно с локальным интерфейсом.

Класс компонента

Классы сеансовых и объектных компонентов фактически реализуют прикладные методы и методы жизненного цикла компонента. Заметьте, что класс компонента для сеансовых и объектных компонентов не реализует непосредственно ни один интерфейс компонента. Однако он должен иметь методы, совпадающие с сигнатурами методов, определенных в удаленных и локальных интерфейсах, и должен иметь методы, соответствующие некоторым методам

в обоих, удаленном и локальном, внутренних интерфейсах. Это звучит совершенно запутанно. В книге все это будет проясняться по мере нашего продвижения вперед. Объектные компоненты должны реализовывать интерфейс `javax.ejb.EntityBean`, а сеансовые компоненты – интерфейс `javax.ejb.SessionBean`. Оба интерфейса – `EntityBean` и `SessionBean` – расширяют `javax.ejb.EnterpriseBean`.

Компоненты, управляемые сообщениями, не используют ни один из интерфейсов компонентов, потому что к ним никогда не осуществляется доступ посредством вызовов методов из других приложений или компонентов. Вместо этого компоненты, управляемые сообщениями, содержат единственный метод `onMessage()`, вызываемый контейнером в случае прибытия нового сообщения. У компонента, управляемого сообщениями, нет компонентных интерфейсов, которые есть у сеансовых и объектных компонентов. Для своей работы ему нужен только один класс компонента. Компонент, управляемый сообщениями, реализует интерфейсы `javax.ejb.MessageDrivenBean` и `javax.ejb.MessageListener`. Именно интерфейс JMS `MessageListener`, а не другие протоколы доставки сообщений, делает компонент, управляемый сообщениями, специфичным для JMS. EJB 2.0 требует использования JMS, но будущие версии могут разрешить применение других систем доставки сообщений. Интерфейс `MessageDrivenBean`, так же как и `EntityBean` с `SessionBean`, расширяет интерфейс `javax.ejb.EnterpriseBean`.

Первичный ключ

Первичный ключ – это очень простой класс, предоставляющий указатель внутри базы данных. Первичный ключ необходим только объектным компонентам. Единственное требование для этого класса состоит в том, чтобы он реализовывал интерфейс `java.io.Serializable`.

В EJB 2.0 определяется важное различие между удаленными и локальными интерфейсами. Локальные интерфейсы предоставляют способ эффективного взаимодействия компонентам, находящимся в одном и том же контейнере. Вызовы методов локального интерфейса не используют RMI. Для методов локального интерфейса не нужно объявлять, что они генерируют `RemoteException`. От компонента не требуется предоставления локального интерфейса, если во время разработки известно, что он будет взаимодействовать только с удаленными клиентами. И наоборот, нет необходимости предоставлять компоненту удаленный интерфейс, если известно, что он будет вызываться только компонентами, расположенными в том же самом контейнере. Можно предоставить либо локальный, либо удаленный, либо оба интерфейса.

Сложности – в частности путаница с классами, реализующими методы какого-либо интерфейса, но не реализующими сам интерфейс, – происходят из-за того, что компоненты расположены между клиент-

ским программным обеспечением и базой данных. Клиент никогда не взаимодействует напрямую с классом компонента, для выполнения своей работы он всегда обращается к методам интерфейса сеансового или объектного компонента, взаимодействуя с автоматически сгенерированной заглушкой. (В этом отношении компонент, нуждающийся в обслуживании другим компонентом, является просто еще одним клиентом – он пользуется теми же заглушками, а не взаимодействует с классом компонента напрямую.)

Хотя в EJB 2.0 локальные интерфейсы компонентов (локальный и локальный внутренний) представляют сеансовые и объектные компоненты, находящиеся в том же адресном пространстве, и не используют протоколы распределенных объектов, они все равно представляют заглушку или заместитель класса компонента. Хотя между совмещенными компонентами не существует сетевых соединений, заглушки позволяют контейнеру наблюдать за взаимодействием между компонентами и при необходимости использовать управление транзакциями и безопасностью.

Важно отметить, что у компонентов, управляемых сообщениями, в EJB 2.0 нет никаких компонентных интерфейсов, но они могут быть клиентами любых сеансовых или объектных компонентов и взаимодействовать с этими компонентами через их интерфейсы. Объектные и сеансовые компоненты, с которыми взаимодействует компонент, управляемый сообщениями, могут быть совмещенными, в этом случае он работает с их локальными компонентными интерфейсами, или они могут располагаться в разных адресных пространствах или контейнерах EJB, – в этом случае используется удаленный интерфейс.

Между компонентом и сервером также происходит большое количество взаимодействий. Эти взаимодействия управляются *контейнером* (*container*), отвечающим за предоставление унифицированного интерфейса между компонентом и сервером. (Многие считают термины «контейнер» и «сервер» взаимозаменяемыми, и это понятно, т. к. различия между ними определены не четко.) Контейнер отвечает за создание нового экземпляра компонента, правильное хранение его сервером и т. д. Инструментальные средства,[1] предоставляемые производителем контейнера, выполняют за кулисами огромный объем работы. По крайней мере одно средство заботится о создании связей между объектными компонентами и записями в базе данных. Другие средства генерируют код, опираясь на интерфейсы компонентов и сам класс компонента. Этот сгенерированный код занимается созданием компонента, хранением его в базе данных и т. д. Этот код (в дополнение к заглушкам) фактически реализует интерфейсы компонента, по каковой причине класс компонента не должен делать этого.

[1] Правильнее в данном случае было бы сказать «службы» или «сервисы». – *Примеч. науч. ред.*

Соглашения по именам

Прежде чем продолжить, давайте заключим несколько соглашений. Говоря о компоненте как о целом – его интерфейсах, классе и т. д., мы будем называть его общим прикладным именем с добавлением сокращения «EJB». Например, компонент, разрабатываемый для моделирования каюты корабля, будет называться «CabinEJB». Заметьте, что мы не используем моноширинный шрифт для слова «Cabin». Это делается из-за того, что мы ссылаемся на все составные части компонента (интерфейсы, класс и т. д.) как на целое, а не просто на одну отдельную часть, такую как удаленный интерфейс или класс компонента. Термин *компонент* (*enterprise bean*) описывает любой тип компонента, например объектный компонент, сеансовый или управляемый сообщениями. *Объектный компонент* (*entity bean*) – это компонент, содержащий данные, *сеансовый компонент* (*session bean*) означает компонент сеансового типа, а *компонент, управляемый сообщениями* (*message-driven bean*), означает тип компонента, управляемого сообщениями. Общеупотребительным является сокращение EJB для обозначения компонента, именно оно используется в данной книге.

Мы также будем использовать суффиксы для различения локальных и удаленных интерфейсов компонентов. Говоря об удаленном интерфейсе компонента Cabin, будем присоединять к общему прикладному имени слово «Remote». Например, удаленный интерфейс компонента Cabin назван интерфейсом `CabinRemote`. В EJB 2.0 локальный интерфейс компонента Cabin будет называться `CabinLocal`. Внутренний интерфейс следует нашему соглашению, добавляя слово «Home» к этому сочетанию. Удаленный и локальный внутренний интерфейсы компонента Cabin будут называться `CabinHomeRemote` и `CabinHomeLocal` соответственно. Класс компонента всегда именуются общим прикладным именем с добавлением слова «Bean». Например, класс компонента Cabin будет называться `CabinBean`.

Удаленный интерфейс

Определив механизм, посмотрим, как строятся объектные компоненты (entity beans) или компоненты с состоянием (stateful enterprise beans) с удаленными интерфейсами. В этом разделе мы рассмотрим компонент Cabin – объектный компонент, моделирующий каюту корабля. Начнем с его удаленного интерфейса.

Определим удаленный интерфейс компонента Cabin с помощью интерфейса `CabinRemote`, определяющего прикладные методы для работы с каютами. Все виды удаленных интерфейсов расширяют интерфейс `javax.ejb.EJBObject`:

```
import java.rmi.RemoteException;
public interface CabinRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String str) throws RemoteException;
```

```
    public int getDeckLevel() throws RemoteException;
    public void setDeckLevel(int level) throws RemoteException;
}
```

Здесь присутствуют методы для задания названия каюты и методы для задания уровня палубы каюты. Наверное, можно придумать массу других методов, которые пригодятся впоследствии, но для начала достаточно и этих. Все эти методы объявляют, что они генерируют исключение `RemoteException`, требуемое для всех методов удаленных компонентных интерфейсов, но не для локальных интерфейсов в EJB 2.0. EJB требует, чтобы для удаленных компонентных интерфейсов использовались соглашения Java RMI-IIOP, хотя базовым протоколом может быть CORBA IIOP, Java Remote Method Protocol (JRMP) или другой. Java RMI-IIOP детально обсуждается в следующей главе.

Удаленный внутренний интерфейс

Удаленный внутренний интерфейс определяет методы жизненного цикла, используемые клиентами объектных и сеансовых компонентов для поиска компонентов. Удаленный внутренний интерфейс расширяет `javax.ejb.EJBHome`. Назовем внутренний интерфейс компонента Cabin `CabinHomeRemote` и определим его так:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;
    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

Метод `create()` отвечает за инициализацию экземпляра нашего компонента. Если вашему приложению будет нужно, вы можете предоставить другие методы `create()` с другими аргументами.

В дополнение к методу `findByPrimaryKey()` можно определить другие методы, предоставляющие удобный способ поиска компонента Cabin. Например, вам может понадобиться определить метод с именем `findByShip()`, возвращающий все каюты отдельного корабля. Поисковые методы, подобные этим, используются в объектных компонентах, но не в сеансовых и, очевидно, что не в компонентах, управляемых сообщениями.

EJB 2.0: класс компонента

А теперь посмотрим на настоящий компонент. Здесь приведен код для компонента `CabinBean`. Это не полная реализация, но она поясняет, как собрать все части вместе:

```
import javax.ejb.EntityContext;

public abstract class CabinBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
        // ничего не делаем
    }

    public abstract String getName();
    public abstract void setName(String str);

    public abstract int getDeckLevel();
    public abstract void setDeckLevel(int level);

    public abstract Integer getId();
    public abstract void setId(Integer id);

    public void setEntityContext(EntityContext ctx){
        // не реализован
    }
    public void unsetEntityContext(){
        // не реализован
    }
    public void ejbActivate(){
        // не реализован
    }
    public void ejbPassivate(){
        // не реализован
    }
    public void ejbLoad(){
        // не реализован
    }
    public void ejbStore(){
        // не реализован
    }
    public void ejbRemove(){
        // не реализован
    }
}
```

Обратите внимание, что класс `CabinBean` объявлен как абстрактный из-за нескольких его методов, считывающих и обновляющих постоянное состояние компонента. Также заметьте, что отсутствуют поля экземпляра, хранящие информацию о состоянии, к которой обращаются данные методы. Это сделано потому, что мы работаем с компонентом, управляемым контейнером, у которого есть свои абстрактные методы, автоматически реализованные контейнерной системой. Это подробно объясняется далее в книге. Объектные компоненты, управляемые

контейнером, в EJB 2.0 являются единственным типом компонентов, которые объявляются как абстрактные с абстрактными методами доступа. Вы не увидите абстрактных классов и методов в других типах объектных компонентов, сеансовых компонентах и компонентах, управляемых сообщениями.

EJB 1.1: класс компонента

Далее приведен код класса `CabinBean` в EJB 1.1.

```
import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public Integer id;
    public String name;
    public int deckLevel;

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
        // нечего не делаем
    }

    public String getName(){
        return name;
    }
    public void setName(String str){
        name = str;
    }

    public int getDeckLevel(){
        return deckLevel;
    }
    public void setDeckLevel(int level){
        deckLevel = level;
    }
    public Integer getId(){
        return id;
    }
    public void setId(Integer id){
        this.id = id;
    }
    public void setEntityContext(EntityContext ctx){
        // не реализован
    }
    public void unsetEntityContext(){
        // не реализован
    }
    public void ejbActivate(){
```

```
        // не реализован
    }
    public void ejbPassivate(){
        // не реализован
    }
    public void ejbLoad(){
        // не реализован
    }
    public void ejbStore(){
        // не реализован
    }
    public void ejbRemove(){
        // не реализован
    }
}
```

EJB 2.0 и 1.1: класс компонента

Методы set и get для имени каюты и уровня палубы являются прикладными методами класса `CabinBean`. Они соответствуют прикладным методам, определенным в удаленном интерфейсе компонента `CabinRemote`. Класс `CabinBean` имеет состояние и поведение, моделирующее понятие каюты. Эти прикладные методы – единственные, видимые клиентским приложениям. Другие методы видны только контейнеру компонентов и самому классу компонента. Например, методы `setId()` и `getId()` определены в классе, но не определены в удаленном интерфейсе. Это означает, что они не могут быть вызваны клиентом этого объектного компонента. Другие методы требуются для компонентной модели EJB и не являются частью открытого (public) объявления класса компонента.

Методы `ejbCreate()` и `ejbPostCreate()` инициализируют экземпляр класса компонента, когда в базу данных добавляется новая запись, соответствующая каюте. Последние семь методов класса `CabinBean` определены в интерфейсе `javax.ejb.EntityBean`. Эти методы представляют собой методы обратного вызова, отвечающие за жизненный цикл экземпляра. Контейнер компонентов вызывает эти методы класса, когда происходят важные события жизненного цикла. Метод `ejbRemove()`, например, информирует объектный компонент о том, что его данные скоро будут удалены из базы данных. Методы `ejbLoad()` и `ejbStore()` сообщают экземпляру компонента о том, что он находится в состоянии чтения или записи в базу данных. Методы `ejbActivate()` и `ejbPassivate()` извещают экземпляр компонента о том, что он будет активирован или деактивирован (процесс освобождения памяти и других ресурсов). Метод `setEntityContext()` предоставляет компоненту интерфейс с контейнером, позволяющий классу компонента получать информацию о самом себе и своем окружении. Метод `unsetEntityContext()` вызывается контейнером и уведомляет экземпляр компонента о том, что ско-

ро ссылка на него будет удалена и он подвергнется процессу сборки мусора.

Все эти методы обратного вызова предоставляют классу компонента *уведомления* (*notifications*) со стороны сервера перед выполнением какого-либо действия либо после него. Эти уведомления просто информируют компонент о событии. Компонент не обязан что-либо с ними делать. Уведомления обратного вызова сообщают компоненту, на каком этапе своего жизненного цикла он находится, когда его собираются загружать, удалять, деактивировать и т. д. Большинство методов обратного вызова имеют отношение к сохранению состояния, которое должно автоматически реализовываться контейнером, а не классом компонента. По той причине, что эти методы определены в интерфейсе `javax.ejb.EntityBean`, класс объектного компонента обязан их реализовать, но от него не требуется делать в этих методах что-либо значительное, если нет такой необходимости. Наш компонент `CabinBean` не должен ничего делать, когда эти методы вызываются, поэтому они содержат пустые реализации. Подробности об этих методах обратного вызова (когда они вызываются и как на них должен реагировать компонент) обсуждаются в главе 11.

Первичный ключ

Первичный ключ – это указатель, хранящий информацию о расположении данных, описывающий уникальную запись или элемент базы данных. С его помощью в методе `FindByPrimaryKey()` внутреннего интерфейса отыскивается указанный элемент. Первичные ключи определяются разработчиком компонента и обязаны иметь такой же тип, как и сериализуемый объект. Компонент Cabin применяет в качестве простого ключа простой тип `java.lang.Integer`. Можно также определять пользовательские первичные ключи, называемые *составными первичными ключами* (*compound primary key*), представляющие собой сложные первичные ключи, состоящие из нескольких различных полей. Первичные ключи подробно описаны в главе 11.

Как насчет сеансовых компонентов?

`CabinBean` – это объектный компонент, но сеансовый компонент был бы чем-то совсем другим. Он расширял бы интерфейс `SessionBean` вместо `EntityBean` и содержал бы метод `ejbCreate()`, инициализирующий состояние компонента без метода `ejbPostCreate()`. У сеансовых компонентов нет ни метода `ejbLoad()`, ни `ejbStore()` из-за того, что они не сохраняют состояние в базе данных. Хотя у сеансовых компонентов есть метод `setSessionContext()`, у них нет метода `unsetSessionContext()`. У сеансовых компонентов есть и метод `ejbActivate()`, и метод `ejbPassivate()`, которые используются сеансовыми компонентами с состоянием для управления состоянием взаимодействия. И наконец, сеансовые компоненты предоставляют метод `ejbRemove()`, вызываемый для извеще-

ния компонентов о том, что они больше не нужны клиенту. Однако этот метод не сообщает компоненту о том, что его данные будут удалены из базы данных, т. к. сеансовый компонент не представляет никаких данных в базе данных.

У сеансовых компонентов нет первичных ключей. Это сделано потому, что сеансовые компоненты сами не являются постоянными, поэтому нет надобности в каком-либо ключе, проецирующемся на базу данных. Сеансовые компоненты детально описаны в главе 12.

EJB 2.0: Как насчет компонентов, управляемых сообщениями?

Для компонентов, управляемых сообщениями, не определяются ни удаленные, ни локальные, ни внутренние интерфейсы, потому что у этих компонентов нет компонентных интерфейсов. Взамен в этих компонентах определены лишь несколько методов обратных вызовов и ни одного прикладного метода. К методам обратного вызова относятся метод `ejbCreate()`, вызываемый во время создания компонента класса, метод `ejbRemove`, вызываемый перед удалением компонента из системы (обычно тогда, когда он больше не нужен контейнеру), методы `setMessageDrivenBeanContext()` и `onMessage()`. Метод `onMessage()` вызывается каждый раз, когда компоненту приходит асинхронное сообщение. Компонент, управляемый сообщениями, не определяет методы `ejbPassivate()`/`ejbActivate()` и `ejbLoad()`/`ejbStore`, потому что они ему не нужны.

У компонентов, управляемых сообщениями, нет первичных ключей по той же причине, что и у сеансовых. У них нет постоянства (они не сохраняют состояние в базе данных), и поэтому нет необходимости в ключе для базы данных. Компоненты, управляемые сообщениями, подробно рассматриваются в главе 13.

Дескрипторы развертывания и файлы JAR

Большая часть информации о том, как происходит управление компонентами во время выполнения, не относится к интерфейсам и классам, обсужденным ранее. Читатели могли заметить, например, что мы не говорили о том, как компоненты взаимодействуют со службой безопасности, транзакциями, системой имен и другими общими для систем распределенных объектов сервисами. Как вы узнали из предыдущего обсуждения, эти типы основных сервисов управляются контейнером компонентов автоматически, но контейнеру все же необходимо знать, каким образом применять эти основные службы к каждому компоненту во время выполнения. Для того чтобы сообщить ему это, мы используем *дескрипторы развертывания* (*deployment descriptor*).

Дескрипторы развертывания служат почти тем же целям, что и файлы свойств. Они позволяют нам настраивать поведение программного продукта (компонента) во время выполнения без необходимости ме-

нять само программное обеспечение. Файлы свойств обычно используются приложениями, а дескрипторы развертывания специфичны для компонентов. Дескрипторы развертывания по своему назначению также похожи на окна свойств (property sheets), используемые в Visual Basic и PowerBuilder. Так же как окна свойств позволяют нам описывать атрибуты времени выполнения визуальных элементов (фоновый цвет, размер шрифта и т. д.), дескрипторы развертывания позволяют нам описывать атрибуты времени выполнения серверных компонентов (безопасность, контекст транзакций и т. п.). Дескрипторы развертывания позволяют настроить определенное поведение компонентов без модификации класса компонента и его интерфейсов.

Когда класс компонента и его интерфейсы уже определены, создается дескриптор развертывания и заполняется информацией о компоненте. Часто интегрированные среды разработки (Integrated Development Environment, IDE) с поддержкой Enterprise JavaBeans позволяют разработчикам графически создавать дескрипторы развертывания с помощью визуальных утилит, подобных окнам свойств. После того как разработчик установил все свойства компонента, дескриптор развертывания сохраняется в файле. Когда дескриптор развертывания создан и записан в файл, компонент может быть упакован в файл JAR с целью дальнейшего развертывания.

Файлы JAR (Java ARchive) – это файлы ZIP, специально используемые для упаковки классов Java (и других ресурсов, таких как изображения), готовых к использованию в приложении. Файлы JAR применяются для упаковки апплетов, приложений Java, компонентов JavaBeans, веб-приложений (сервлетов и страниц JSP) и компонентов Enterprise JavaBeans. Файл JAR содержит один или несколько компонентов, включая классы, интерфейсы и вспомогательные классы для каждого компонента. Он также содержит один дескриптор развертывания, используемый всеми компонентами файла JAR. Во время развертывания компонента путь к файлу JAR передается средству развертывания контейнера, считывающему этот файл.

Во время развертывания, когда считывается файл JAR, инструментальные средства контейнера загружают дескриптор развертывания для получения информации о компоненте и о том, как управлять им во время выполнения. Дескриптор развертывания сообщает средствам развертывания типы компонентов, хранящихся в файле JAR (сеансовые, объектные или управляемые сообщениями), как ими следует управлять в транзакциях, у кого есть доступ к компонентам во время выполнения, а также другие атрибуты времени выполнения компонентов. Лицо, развертывающее компонент, может изменить некоторые из этих установок, например атрибуты безопасности и транзакций, для настройки компонента под конкретное приложение. Многие инструментальные средства контейнеров предоставляют окна свойств для графического изображения и изменения дескриптора развертывания

во время развертывания компонента. Эти графические окна свойств похожи на те, которыми пользуются разработчики компонентов.

Дескрипторы развертывания помогают инструментам развертывания размещать компоненты в контейнер компонентов. Когда компонент развернут, свойства, описанные в дескрипторе развертывания, продолжают использоваться контейнером компонентов для управления компонентами во время выполнения.

В версии 1.0 Enterprise JavaBeans в качестве дескрипторов развертывания выступали сериализуемые классы дескрипторов развертывания. Начиная с Enterprise JavaBeans версии 1.1 сериализуемые классы, используемые в версии 1.0, были заменены более гибким файловым форматом, опирающимся на расширенный язык разметки (eXtensible Markup Language, XML). XML-дескрипторы развертывания – это текстовые файлы, структурированные в соответствии со стандартным для EJB «дескриптором» – определением типа документа (Document Type Definition, DTD), который может быть расширен. Поэтому тип информации для развертывания, хранящейся в нем, может развиваться вместе со спецификацией. В главе 16 дается подробное описание XML-дескриптора развертывания. В следующих разделах дается краткий обзор XML-дескриптора развертывания.

EJB 2.0: дескриптор развертывания

Следующий дескриптор развертывания мог бы быть использован в EJB 2.0 для описания компонента Cabin.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD EnterpriseJavaBeans
2.0//EN" "http:// java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.CabinHomeRemote</home>
            <remote>com.titan.CabinRemote</remote>
            <local-home>com.titan.CabinHomeLocal</local-home>
            <local>com.titan.CabinLocal</local>
            <ejb-class>com.titan.CabinBean </ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

EJB 1.1: дескриптор развертывания

Следующий дескриптор развертывания мог бы быть использован в EJB 1.1 для компонента Cabin.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD EnterpriseJavaBeans
2.0//EN" "http:// java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.CabinHomeRemote</home>
            <remote>com.titan.CabinRemote</remote>
            <local-home>com.titan.CabinHomeLocal</local-home>
            <local>com.titan.CabinLocal</local>
            <ejb-class>com.titan.CabinBean </ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

EJB 2.0 и 1.1: элементы XML дескриптора развертывания

В дескрипторе развертывания для реального компонента должно быть гораздо больше информации. Этот пример просто иллюстрирует виды информации, которые вам попадутся в XML-дескрипторе развертывания.

Первый элемент любого XML-документа – это `<!DOCTYPE>`. Этот элемент описывает структуру данного XML-документа, определенную в DTD, версию DTD и URL, по которому расположен DTD. DTD описывает способ, которым структурируется отдельный XML-документ.

Все остальные элементы XML-документа специфичны для EJB. Здесь представлены не все элементы, применяемые в дескрипторах развертывания, а просто показаны типы элементов. Далее поясняется, что означают эти элементы.

`<ejb-jar>`

Корневой элемент XML дескриптора развертывания. Все остальные элементы должны быть вложены в него. А он, в свою очередь, должен содержать один элемент `<enterprise-beans>` и может содержать другие дополнительные элементы.

`<enterprise-beans>`

Содержит объявления всех компонентов, описываемых этим XML-документом. Он может содержать элементы `<entity>`, `<session>` и `<message-driven>`, описывающие соответственно объектные, сеансовые компоненты и компоненты, управляемые сообщениями.

`<entity>`

Описывает объектный компонент и содержит информацию о его развертывании. На каждый объектный компонент, описываемый этим XML-документом, должен приходиться один такой элемент.

Элемент <session> используется таким же образом для описания сеансового компонента. Элемент <message-driven> несколько отличается, поскольку не определяет ни один компонентный интерфейс.

<ejb-name>

Имя, описывающее компонент. Это то имя компонента, которое мы используем в разговоре, когда говорим о компоненте в целом.

<home>

Полное имя класса удаленного внутреннего интерфейса. Этот интерфейс определяет поведение, связанное с жизненным циклом (создание, поиск, удаление) компонента по отношению к его клиентам, находящимся вне контейнерной системы.

<remote>

Полное имя класса удаленного интерфейса. Этот интерфейс определяет прикладные методы компонента со стороны его клиентов, расположенных за пределами контейнера.

EJB 2.0: <local-home>

Полное имя класса локального внутреннего интерфейса. Этот интерфейс определяет поведение компонента, связанное с жизненным циклом (создание, поиск, удаление), по отношению к другим совмещенным с ним компонентам.

EJB 2.0: <local>

Полное имя класса локального интерфейса. Этот интерфейс определяет прикладные методы компонента со стороны совмещенных с ним компонентов.

<ejb-class>

Полное имя класса компонента. Этот класс реализует прикладные методы компонента.

<prim-key-class>

Полное имя класса первичного ключа компонента. Первичный ключ используется для поиска данных компонента в базе данных.

Последние два компонента в дескрипторе развертывания, <persistence-type> и <reentrant>, определяют стратегию постоянства и политику параллелизма объектного компонента. Эти элементы объясняются более подробно далее в этой книге.

По мере продвижения вперед мы рассмотрим элементы, описывающие понятия, которых мы еще не касались, поэтому сейчас не имеет смысла стремиться узнать все обо всех элементах, которые могут встретиться в дескрипторе развертывания.

Компонентный и внутренний объекты

Объектные и сеансовые компоненты объявляют компонентные интерфейсы, применяемые для доступа к ним их клиентами. В EJB 2.0 и 1.1 клиенты, находящиеся за пределами контейнерной системы, такие как сервлеты или приложения Java, всегда будут использовать удаленный интерфейс компонента. В EJB 2.0 клиенты, являющиеся клиентами в той же контейнерной системе, чаще всего осуществляют взаимодействие посредством локального интерфейса компонента. В этом разделе объясняется, как интерфейсы компонента во время выполнения взаимодействуют с классом компонента.

Это обсуждение поможет нам разобраться с объектными и сеансовыми компонентами, но оно вовсе не относится к компонентам, управляемым сообщениями, поскольку компоненты эти не объявляют компонентных интерфейсов. Компоненты, управляемые сообщениями, – звери совсем иной породы, поэтому полное их описание отложим до главы 13.

Итак, у вас уже есть базовое понимание некоторых отдельных частей компонентов (интерфейсы, класс компонента и дескриптор развертывания), и сейчас самое время поговорить немного подробнее о том, как эти части собираются вместе внутри контейнерной системы EJB. К сожалению, нам не удастся поговорить так подробно, как хотелось бы. У контейнера компонентов есть несколько способов реализации этих отношений. Мы покажем некоторые из этих возможностей. А именно, поговорим о том, как реализация контейнером интерфейсов объектных и сеансовых компонентов позволяет клиентам, приложениям, находящимся за пределами контейнера, или другим совмещенным компонентам взаимодействовать с классом компонентов и вызывать его методы.

Две пропущенных части – это сам компонентный (EJB object) и внутренний объекты (EJB home). Очень может быть, что вы никогда не встретитесь с этими классами, т. к. их определения являются собственными расширениями реализации конкретного производителя и, как правило, не делаются общедоступными. И это хорошо, поскольку это не что иное, как разделение ответственности в соответствии с областями компетентности (каждый должен заниматься своим делом). Разработчик приложения хорошо знаком с тем, как работает прикладная среда и как нужно ее моделировать, поэтому он будет концентрироваться на создании приложений и компонентов, описывающих его прикладную задачу. Разработчики системного уровня – те люди, которые пишут серверы EJB, не понимают прикладную задачу, но они знают, как разработать СТМ и поддерживать распределенные объекты. Разработчикам системного уровня имеет смысл приложить свои знания к созданию механизма управления распределенными объектами, а прикладную логику оставить разработчику приложения. Давайте

поговорим о компонентном и внутреннем объектах – необходимо, чтобы вы увидели эти пропущенные фрагменты общей картины.

Компонентный объект

В данной главе много говорилось об удаленных и локальных интерфейсах компонента, расширяющих, соответственно, интерфейсы `EJBObject` и, для EJB 2.0, `EJBLocalObject`. Кто реализует эти интерфейсы? Ясно, что заглушка: мы это хорошо знаем. Но что же на стороне сервера?

А на стороне сервера именно компонентный объект и реализует удаленный и/или локальный (EJB 2.0) интерфейсы компонента. Компонентный объект заключает в себя экземпляр компонента, т. е. экземпляр класса компонента, созданный вами (в нашем примере это `CabinBean`) и находящийся на сервере, и расширяет его возможности с помощью добавления функциональности классов `javax.ejb.EJBObject` и/или `javax.ejb.EJBLocalObject`.

В дальнейшем, обсуждая, какой именно интерфейс реализуется компонентным объектом, мы будем довольно часто говорить «и/или». Дело в том, что компоненты в EJB 2.0 могут объявить локальный или удаленный интерфейс, или оба. Локальные интерфейсы не использовались в EJB 1.1, поэтому, если вы работаете с этой версией, пропускайте ссылки на них, они имеют значение только для контейнерных систем EJB 2.0.

В EJB 2.0, независимо от того, какой интерфейс реализует компонент, мы можем считать, что компонентный объект реализует оба интерфейса. В действительности может существовать специальный компонент для удаленного интерфейса каждого компонента и другой специальный компонент – для локального интерфейса каждого компонента. Что зависит от способа, каким производитель решил это реализовать. Для наших целей термин «компонентный объект» будет использоваться в обсуждении и локального, и удаленного интерфейсов, и обоих сразу. Функциональные возможности этих интерфейсов с точки зрения компонентного объекта настолько похожи, что нет необходимости обсуждать отдельно каждый компонентный объект.

Компонентный объект генерируется утилитами, поставляемыми вместе с контейнерами EJB, на основе классов компонентов и информации, расположенной в дескрипторе развертывания. Компонентный объект заключает в себе экземпляр и вместе с контейнером обеспечивает транзакции, безопасность и другие системные операции компонента времени выполнения. В главе 3 более подробно говорится о роли компонентного объекта в операциях системного уровня.

Производитель может использовать ряд подходов для реализации компонентного объекта. На рис. 2.1 показаны три способа работы с ин-

терфейсом CabinBean. Такие же стратегии реализации применяются и к интерфейсам CabinLocal и javax.ejb.EJBLocalObject.

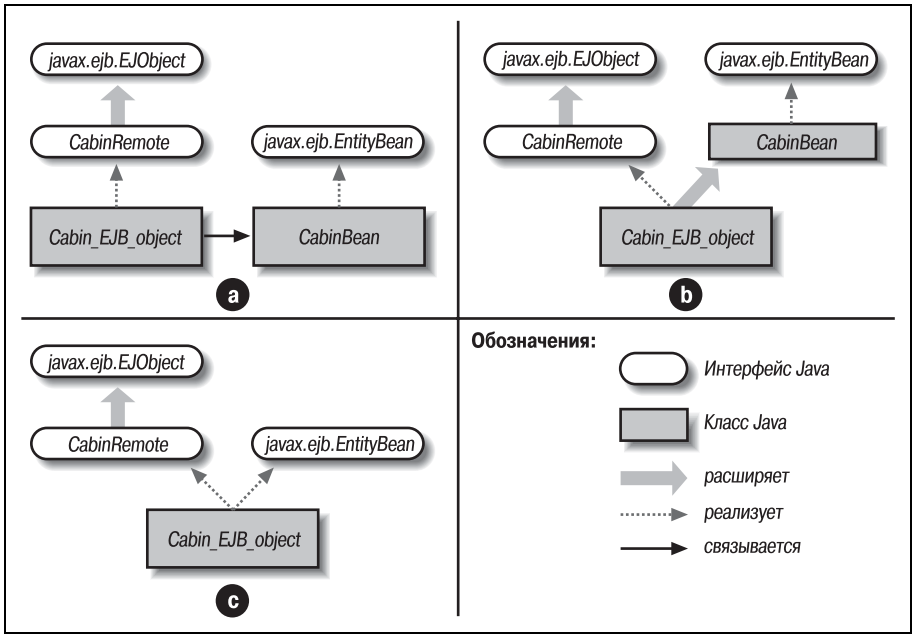

Рис. 2.1. Три способа реализации компонентного объекта

На рис. 2.1(a) компонентный объект представляет собой классическую «обертку», поскольку хранит ссылку на класс компонента и перенаправляет ему все запросы. На рис. 2.1(b) показано, как класс компонентного объекта в действительности расширяет класс компонента, добавляя функциональность, специфичную для его контейнера компонентов. На рис. 2.1(c) класс компонента уже не включен в модель. В этом случае компонентный объект содержит как собственную реализацию, требуемую контейнером компонентов, так и реализацию методов прикладного компонента, скопированную из его класса.

Дизайн компонентного объекта, показанный на рис. 2.1(a), вероятно, наиболее общеупотребителен. На протяжении этой книги, и особенно в данной главе, мы будем обсуждать функционирование EJB в предположении, что компонентный объект включает в себя класс прикладного компонента, как изображено на рис. 2.1(c). Но могут использоваться и другие реализации. Не должно иметь большого значения, какую из них выбрал ваш производитель. Основной момент состоит в том, что в действительности вам не нужно много знать о компонентном объекте, т. к. его реализация – обязанность производителя. Уже одно то, что мы знаем о существовании этого компонента, снимет много вопросов об устройстве корпоративных компонентов. Информация о компонен-

те, необходимая клиентам (в состав которых входят и другие компоненты), описывается в удаленных и внутренних интерфейсах.

Внутренний объект

Внутренний объект во многом похож на компонентный объект. Он представляет собой еще один класс, автоматически генерируемый во время установки компонента в контейнер. Он реализует все методы, определенные во внутреннем интерфейсе (удаленном и/или локальном), и отвечает за помощь контейнеру в управлении жизненным циклом компонента. Работая вплотную с контейнером компонентов, внутренний объект обеспечивает поиск, создание и удаление компонентов. Для этого может потребоваться взаимодействие с менеджерами ресурсов, пулом экземпляров и механизмами постоянства, детали которых скрыты от разработчика.

Например, при вызове метода `create()` внутреннего интерфейса внутренний объект создает экземпляр компонентного объекта, ссылающегося на экземпляр прикладного компонента подходящего типа. Когда экземпляр компонента связан с компонентным объектом, вызывается соответствующий метод экземпляра `ejbCreate()`. В случае объектного компонента в базу данных вставляется новая запись. При использовании сеансового компонента происходит простая инициализация экземпляра. По завершении метода `ejbCreate()` внутренний объект возвращает клиенту удаленную или локальную ссылку (т. е. заглушку) на компонентный объект. После этого клиент может начать работу с компонентным объектом, вызывая его прикладные методы с помощью заглушки. Заглушка вызывает методы компонентного объекта, который, в свою очередь, перенаправляет эти вызовы экземпляру компонента.

Откуда внутренний объект в EJB 2.0 знает, какие типы ссылок (локальные или удаленные) на компонентный объект надо возвращать? Это зависит от используемого внутреннего интерфейса. Если клиент вызывает метод `create()` удаленного внутреннего интерфейса, внутренний объект вернет ссылку на удаленный интерфейс. Если же клиент работает с локальным внутренним интерфейсом, внутренний объект вернет ссылку, реализующую локальный интерфейс. EJB 2.0 требует, чтобы типом результата методов удаленного внутреннего интерфейса был удаленный интерфейс, а локального внутреннего интерфейса – локальный интерфейс:

```
// Удаленный внутренний интерфейс компонента Cabin
public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;
    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

```
// Локальный внутренний интерфейс компонента Cabin
public interface CabinHomeLocal extends javax.ejb.EJBHome {
    public CabinLocal create(Integer id)
        throws CreateException;
    public CabinLocal findByPrimaryKey(Integer pk)
        throws FinderException;
}
```

На рис. 2.2 показана архитектура EJB с внутренним и компонентным объектами, реализующими соответственно внутренний интерфейс и удаленный или локальный интерфейс. Класс прикладного компонента показан включенным внутрь компонентного объекта.

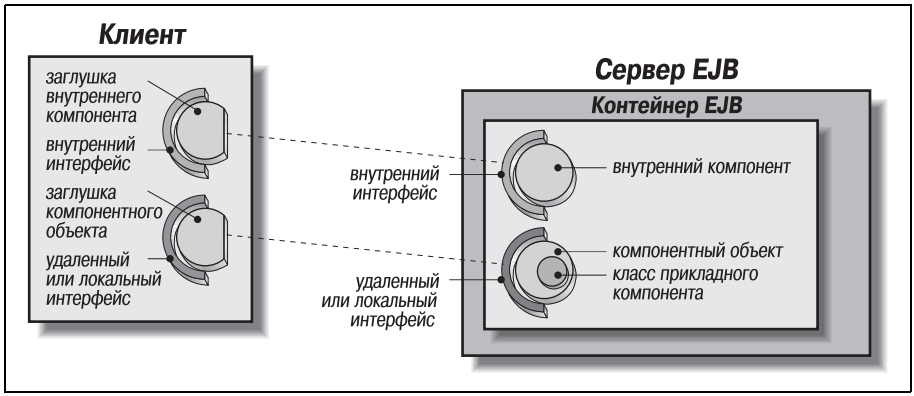

Рис. 2.2. Архитектура EJB

На протяжении этой книги мы будем рассматривать компонентный и внутренний объекты как конструкции, поддерживающие как удаленный, так и локальный интерфейсы компонента. В действительности мы не знаем, как производитель решил реализовать компонентный и внутренний объекты, потому что они представляют собой всего лишь логические конструкции, которые могут и не иметь программных аналогов. Важно помнить, что «компонентный объект» и «внутренний объект» – просто термины, предназначенные для описания обязанностей контейнера компонентов по поддержке интерфейсов компонентов. Мы решили в этой книге дать им более четкое определение только с целью обучения. Реализации компонентного и внутреннего объектов, обсуждаемые здесь, рассматриваются как иллюстрации, а не действительное представление о реализации этих понятий.

Развертывание компонента

Компонентный и внутренний объекты автоматически генерируются во время развертывания. После того как файлы, описывающие компонент (компонентные интерфейсы и классы), будут упакованы в файл JAR, компонент становится готовым к развертыванию, т. е. он может быть размещен в контейнере компонентов и стать доступным в качест-

ве распределенного объекта. Во время развертывания поставляемые вместе с контейнером инструментальные средства генерируют классы компонентного и внутреннего объектов, используя для этого информацию из дескриптора развертывания и других интерфейсов и классов в файле JAR.

Использование компонентов

Давайте посмотрим, как клиент должен работать с компонентом для того, чтобы сделать что-нибудь полезное. Начнем мы с компонента Cabin, определенного ранее. Каюта (cabin) – это место, описание которого хранится в базе данных. Для того чтобы сделать пример несколько более реалистичным, представим себе, что существуют еще и другие компоненты: Ship, Cruise, Ticket, Customer, Employee и т. д.

Получение информации из объектного компонента

Представим себе, что клиенту нужно отобразить графически информацию об отдельном рейсе, такую как название рейса, название корабля и список кают. С помощью идентификатора рейса, полученного из текстового поля, мы можем использовать несколько наших компонентов для заполнения экрана клиента данными по запрошенному рейсу. Код для выполнения всего этого будет выглядеть следующим образом:

```
CruiseHomeRemote cruiseHome = ... используем JNDI для получения внутреннего
объекта
// Получаем из текстового поля идентификатор круиза
String cruiseID = textFields1.getText();
// Создаем из идентификатора первичный ключ компонента
Integer pk = new Integer(cruiseID);
// Используем этот первичный ключ для поиска круиза
CruiseRemote cruise = cruiseHome.findByPrimaryKey(pk);
// Записываем в текстовое поле textField2 название круиза
textField2.setText(cruise.getName());
// Получаем из компонента cruise удаленную ссылку
// на используемое круизом судно
ShipRemote ship = cruise.getShip();
// Записываем в текстовое поле textField3 название судна
textField3.setText(ship.getName());

// Получаем все каюты данного судна
Collection cabins = ship.getCabins();
Iterator cabinItr = cabins.iterator();

// Просматриваем коллекцию, внося названия всех кают
// в элемент список listBox1
while( cabinItr.hasNext())
    CabinRemote cabin = (CabinRemote)cabinItr.next();
    listBox1.addItem(cabin.getName());
}
```

Начнем с получения ссылки на внутренний объект объектного компонента, представляющего круиз. Мы используем удаленный интерфейс вместо локального из-за того, что клиент – это графическое приложение Java, расположенное за пределами контейнера. В EJB 1.1 у нас не будет выбора, поскольку там поддерживаются только удаленные компонентные интерфейсы. В примере не показано, что ссылка на внутренний объект получается с помощью JNDI – мощного программного интерфейса для поиска ресурсов, таких как удаленные объекты, в сети. Несколько сложно обсуждать его сейчас, поэтому он будет рассмотрен в последующих главах.

Мы считываем ID рейса из текстового поля, создаем с его помощью первичный ключ и затем используем этот первичный ключ вместе с внутренним объектом для получения ссылки `CruiseRemote`, реализующей прикладные методы нашего компонента. Получив нужный компонент Cruise, мы можем попросить его дать нам удаленную ссылку на компонент Ship (корабль), соответствующий этому рейсу. Мы можем затем получить коллекцию (`Collection`) ссылок на удаленные компоненты Cabin (каюта) этого компонента Ship и отобразить названия компонентов Cabin для клиента.

Объектные компоненты моделируют данные и поведение. Они предоставляют многократно используемый и последовательный интерфейс к записям в базе данных. Поведение объектных компонентов обычно направлено на применение прикладных правил, имеющих непосредственное отношение к манипуляции с данными. Кроме этого, объектные компоненты могут моделировать отношения с другими элементами. У корабля, например, есть множество кают. Мы можем получить список кают, находящихся на корабле, вызвав метод `ship.getCabins()`.

Моделирование рабочего потока с помощью сеансовых компонентов

Объектные компоненты полезны для хранения данных и описания прикладных понятий, выражаемых при помощи существительных, но они не очень удобны для представления процесса или задачи. Компонент Ship предоставляет методы и поведение для выполнения действий непосредственно над кораблем, но не определяет контекст, в котором эти действия выполняются. В предыдущем примере мы получили данные о рейсах и кораблях. Мы также могли модифицировать эти данные. С небольшим усилием мы также могли бы представить, как регистрировать пассажиров – возможно, путем добавления компонента Customer (пассажир) к компоненту Cruise или включения клиентов в список, поддерживаемый кораблем. Мы могли бы попытаться втиснуть методы для приема оплаты и других задач, связанных с регистрацией, в клиентское приложение или даже в компоненты Ship и Cabin, но это было бы слишком сложным и неподходящим решением. Мы не должны включать прикладную логику в клиентское приложение –

именно поэтому мы поставили многоуровневую архитектуру на первое место. Точно так же, мы не должны включать этот тип логики и в наши объектные компоненты, представляющие корабли и каюты. Регистрация пассажиров и расписание движения кораблей являются действиями или функциями прикладной задачи, а не компонентов Ship и Cabin, и, следовательно, выражаются в терминах процесса или задачи.

Сеансовые компоненты действуют в качестве посредников клиента по управлению прикладными процессами или задачами. Они представляют собой подходящее место для размещения прикладной логики. Сеансовый компонент, в отличие от объектного, не является постоянным. Ни одна из частей сеансового компонента не связана напрямую с базой данных и не сохраняется в промежутках между сессиями. Сеансовые компоненты работают с компонентами, с данными и другими ресурсами для управления *рабочим потоком* (*workflow*). Рабочий поток является ключевым понятием прикладной системы, т. к. он выражает взаимодействия компонентов, необходимые для моделирования реальной прикладной задачи. Сеансовые компоненты управляют задачами и ресурсами, но сами не представляют никаких данных.

Хотя термин «рабочий поток» часто служит для описания управления прикладными процессами, которые могут длиться несколько дней и требовать интенсивного вмешательства человека, в данной книге подразумевается не это значение. Здесь термин «рабочий поток» описывает взаимодействие компонентов внутри одной транзакции, которая может длиться всего несколько секунд.

В следующем коде показано как сеансовый компонент, разработанный для целей резервирования билетов на круиз, может управлять рабочим потоком других сеансовых и объектных компонентов для выполнения своей задачи. Представьте себе, что фрагмент клиентской программы, в нашем случае – пользовательский интерфейс, получает удаленную ссылку на сеансовый компонент TravelAgent. С помощью информации, введенной клиентом в текстовое поле, клиентское приложение вносит одного пассажира в список рейса.

```
// Извлекаем из текстового поля номер кредитной карточки
String creditCard = textField1.getText();
int cabinID = Integer.parseInt(textField2.getText());
int cruiseID = Integer.parseInt(textField3.getText());

// Создаем новую сессию Reservation, передавая в нее
// ссылку на объектный компонент customer
TravelAgent travelAgent = travelAgentHome.create(customer);
// Устанавливаем идентификаторы судна и каюты
travelAgent.setCabinID(cabinID);
travelAgent.setCruiseID(cruiseID);
```

```
// Используя номер карточки и цену, выполняем резервирование
// Этот метод возвращает объект Ticket
TicketDO ticket = travelAgent.bookPassage(creditCard, price);
```

Это достаточно высокоуровневая (грубая) абстракция процесса регистрации пассажира: большинство деталей спрятано от клиента. Сокрытие мелких деталей рабочего потока важно потому, что это дает нам большую гибкость при дальнейшем усовершенствовании системы и в том, как разрешать клиентам взаимодействовать с системой EJB.

В следующем листинге показан фрагмент кода класса `TravelAgentBean`. Метод `bookPassage()` на самом деле работает с тремя объектными компонентами (Customer, Cabin и Cruise) и еще одним сеансовым компонентом – ProcessPayment. Компонент ProcessPayment предоставляет несколько методов для проведения платежа, включающих чеки, наличные и кредитные карты. В данном случае мы используем сессию (сеанс) ProcessPayment для оплаты билета на круиз с помощью кредитной карточки. Как только оплата будет произведена, будет создан и возвращен в клиентское приложение сериализуемый объект `TicketDO`.

```
public class TravelAgentBean implements javax.ejb.SessionBean {
    public CustomerRemote customer;
    public CruiseRemote cruise;
    public CabinRemote cabin;

    public void ejbCreate(CustomerRemote cust){
        customer =cust;

    public TicketDO bookPassage(CreditCardDO card,double price)
        throws IncompleteConversationalState {
        if (customer == null ||cruise == null ||cabin == null){
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeRemote resHome = (ReservationHomeRemote)
                getHome("ReservationHome",ReservationHomeRemote.class);
            ReservationRemote reservation =
                resHome.create(customer,cruise,cabin,price,new Date());
            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
               getHome("ProcessPaymentHome",ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer,card,price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
            return ticket;
        }catch(Exception e){
            throw new EJBException(e);
        }
    }
```

```
// Другие прикладные методы и методы обратного вызова
}
```

В этом примере опущены некоторые детали, однако он хорошо иллюстрирует различие в применении сеансовых и объектных компонентов. Объектные компоненты представляют поведение и данные прикладного объекта, а сеансовые моделируют рабочий поток компонентов. Клиентское приложение использует компонент TravelAgent для выполнения своей задачи с помощью других компонентов. Например, компонент TravelAgent использует при регистрации пассажира компоненты ProcessPayment и Reservation. Компонент ProcessPayment обрабатывает операции с кредитной карточкой, а компонент Reservation выполняет действительное резервирование в системе. Сеансовые компоненты также могут применяться для чтения, обновления и удаления данных, которые не могут быть адекватно отражены объектными компонентами. Сеансовые компоненты, в отличие от объектных, не представляют записи или другую информацию в базе данных, но они все же могут обращаться к данным, находящимся в ней.

Вся работа, выполняемая сеансовым компонентом TravelAgent, могла бы быть реализована в клиентском приложении. Разрешение клиенту напрямую взаимодействовать с объектными компонентами является распространенным, но несколько нелогичным подходом к разработке, т. к. это напрямую привязывает клиента к деталям реализации прикладной задачи. Нелогично это по двум причинам. Во-первых, любое изменение объектных компонентов или их взаимодействий потребует изменений на стороне клиента. И, во-вторых, становится очень сложно повторно использовать код, моделирующий рабочий поток.

Сеансовые компоненты являются достаточно крупными блоками, позволяющими клиенту выполнять его задачи, не заботясь об их деталях. Это дает разработчикам возможность совершенствовать сеансовые компоненты, возможно, изменяя рабочий поток, без влияния на клиентскую программу. Кроме того, если сеансовый компонент разработан должным образом, другие клиенты, выполняющие эту же задачу, могут работать с ним. Сеансовый компонент ProcessPayment, например, может быть использован во многих областях, связанных с резервированием, таких как оптовая и розничная торговля. Например, судовой магазин с подарками может использовать компонент ProcessPayment для обработки покупок. В качестве клиента компонента ProcessPayment компоненту TravelAgent не нужно заботиться о том, как ProcessPayment функционирует. Его интересует только укрупненный интерфейс компонента ProcessPayment, который проверяет и фиксирует платежи.

Перемещение логики рабочего потока в сеансовый компонент помогает также облегчить клиентское приложение и уменьшить сетевой трафик и количество соединений. Переполнение сетевого трафика действительно является одной из серьезных проблем в системах распре-

деленных объектов. Чрезмерный трафик может «потопить» сервер или засорить сеть, увеличить время ответа и уменьшить производительность. Сеансовые компоненты, если их использовать правильно, могут основательно уменьшить сетевой трафик, ограничив число запросов, требуемых для выполнения задачи. В распределенных объектах каждый вызов метода порождает сетевую посылку. Распределенные объекты для передачи запросов используют цикл RMI. А он требует передачи данных между заглушкой и каркасом при каждом вызове метода. В случае применения сеансовых компонентов взаимодействия между компонентами в рабочем потоке происходят на сервере. Один вызов метода в клиентском приложении приводит к многочисленным вызовам на сервере, но через сеть проходит только посылка, порожденная одним вызовом сеансового компонента. В компоненте TravelAgent клиент вызывает метод `bookPassage()` и производит несколько вызовов методов из интерфейсов других компонентов. Выполнив через сеть один вызов, клиент получает несколько вызовов методов.[1]

Кроме этого, сеансовые компоненты уменьшают количество необходимых клиенту сетевых соединений. Затраты на поддержание множественных сетевых соединений могут быть очень высокими, поэтому сокращение числа соединений, требуемых клиенту, повышает производительность системы в целом. Если управление рабочим потоком выполняется с применением сеансовых компонентов, количество соединений с сервером, поддерживаемых каждым клиентом, может быть существенно уменьшено, что приводит к улучшению производительности сервера EJB. На рис. 2.3 сравниваются сетевые трафики и соединения, используемые клиентом, применяющим только объектные компоненты, с теми, которые используются клиентом, применяющим сеансовые компоненты.

Сеансовые компоненты также уменьшают количество используемых клиентом заглушек, что сохраняет память клиента и циклы процессора. Это может показаться не очень важным, но без применения сеансовых компонентов клиент может оказаться вынужденным управлять сотнями или даже тысячами удаленных ссылок одновременно. В компоненте TravelAgent, например, метод `bookPassage()` работает с несколькими удаленными ссылками, тогда как клиенту предоставляется только удаленная ссылка на компонент TravelAgent.

---

[1] В EJB 2.0 метод `bookPassage()` может использовать локальные компонентные интерфейсы других компонентов, поскольку такой подход более эффективен.

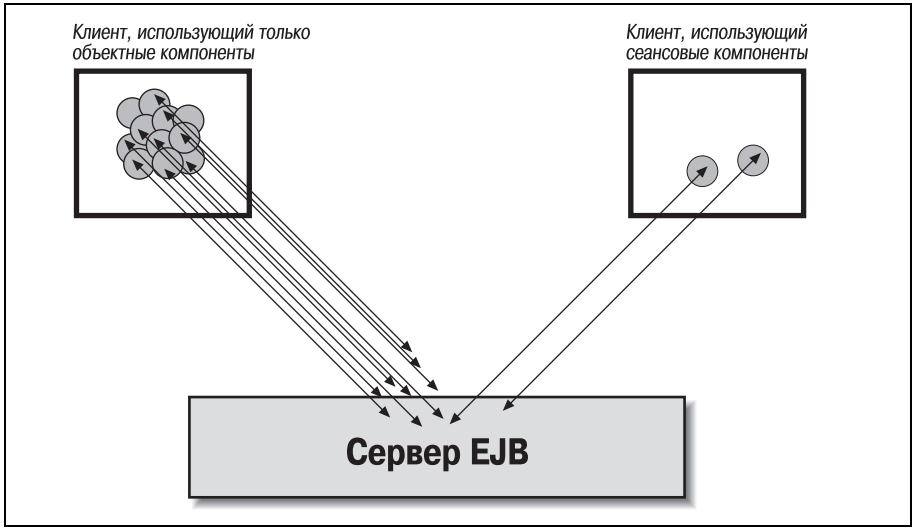

Рис. 2.3. Сеансовые компоненты уменьшают сетевой трафик и облегчают клиент

Сеансовые компоненты с состоянием и без состояния

Сеансовые компоненты могут быть *с состоянием* (*stateful*) и *без состояния* (*stateless*). Компоненты с состоянием при использовании клиентом поддерживают *состояние диалога* (*conversational state*). Состояние диалога не заносится в базу данных, оно сохраняется в памяти все время, пока клиент использует сессию. Поддержание состояния диалога позволяет клиенту осуществлять диалог без объектных компонентов. Состояние сеансового компонента может измениться при вызове любого его метода, и это изменение может привести к последующим вызовам методов. У сеансового компонента TravelAgent, например, может быть гораздо больше методов, а не только `bookPassage()`. Примерами служат методы, устанавливающие идентификаторы каюты или рейса. Эти установочные (set) методы отвечают за изменение состояния диалога. Они преобразуют идентификатор (ID) в удаленные ссылки на компоненты Cabin и Cruise, позже используемые в методе `bookPassage()`. Состояние диалога поддерживается только до тех пор, пока клиентское приложение активно использует компонент. Если клиент отключается или освобождает компонент TravelAgent, состояние диалога теряется навсегда. Сеансовый компонент с состоянием не может использоваться несколькими клиентами. Он выделяется для одного клиента на все время своей жизни.

Сеансовые компоненты без состояния не поддерживают никакого состояния диалога. Каждый метод полностью независим и пользуется только данными, переданными через его параметры. Компонент ProcessPayment – прекрасный пример компонента без состояния. Компо-

ненту ProcessPayment не нужно поддерживать никакое состояние диалога в промежутках между вызовами методов. Вся информация, требуемая для проведения оплаты, передается через метод `byCreditCard()`. Сеансовые компоненты без состояния имеют более высокую производительность, с точки зрения пропускной способности и потребления ресурсов, чем объектные компоненты и сеансовые компоненты с состоянием, поскольку для обслуживания сотен, а возможно и тысяч клиентов требуется всего несколько экземпляров сеансового компонента без состояния. В главе 12 о применении сеансовых компонентов без состояния рассказывается более подробно.

EJB 2.0: Доступ к компонентам с помощью компонентов, управляемых сообщениями

Компоненты, управляемые сообщениями, служат точкой подключения других приложений, желающих взаимодействовать с приложением EJB. Приложение Java или существующие системы (legacy systems), которым нужен доступ к приложению EJB, могут через JMS посылать сообщения компонентам, управляемым сообщениями. Эти компоненты могут затем обработать данные сообщения и выполнить требуемую задачу, используя другие объектные и сеансовые компоненты.

Во многих случаях компоненты, управляемые сообщениями, играют ту же роль, что и сеансовые компоненты, управляя рабочим потоком объектных и сеансовых компонентов для выполнения поставленной задачи. Задача, требующая выполнения, начинается с посылки приложением асинхронного сообщения с помощью системы JMS. В отличие от сеансовых компонентов, отвечающих на вызовы прикладных методов своего интерфейса, компоненты, управляемые сообщениями, реагируют на асинхронные сообщения, приходящие благодаря их методу `onMessage()`. Поскольку сообщения являются асинхронными, клиент, посылающий их, не будет ожидать ответа. Клиенты просто посылают сообщения и забывают о них.

В качестве примера мы можем переделать разработанный ранее компонент TravelAgent в компонент, управляемый сообщениями, ReservationProcessor:

```
public class ReservationProcessorBean implements
javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener {

    public void onMessage(Message message) {
        try {
            MapMessage reservationMsg = (MapMessage)message;

            Integer customerPk =
(Integer)reservationMsg.getObject("CustomerID");
           Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
```

```
            Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");
            double price = reservationMsg.getDouble("Price");

            CreditCardDO card = getCreditCard(reservationMsg);
            CustomerRemote customer = getCustomer(customerPk);
            CruiseLocal cruise = getCruise(cruisePk);
            CabinLocal cabin = getCabin(cabinPk);

            ReservationHomeLocal resHome = (ReservationHomeLocal)
                jndiContext.lookup("java:comp/env/ejb/ReservationHome");
            ReservationLocal reservation =
                resHome.create(customer,cruise,cabin,price,new Date());

            Object ref = jndiContext.lookup("java:comp/env/ejb/
                                            ProcessPaymentHome");
            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer,card,price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
// Другие вспомогательные методы и методы обратного вызова
}
```

Заметьте, что вся информация о резервировании извлекается из сообщения, оставленного компоненту. Сообщения JMS могут принимать разнообразные формы, одна из которых – `javax.jms.MapMessage` – используется в этом примере и несет пару «имя-значение». Как только информация получена из сообщения и извлечены ссылки на компоненты, процесс резервирования протекает точно так же, как и в сеансовом компоненте. Единственное отличие состоит в том, что `TicketDO` не возвращается вызывающему объекту. Компонент, управляемый сообщениями, не долен отвечать ему из-за того, что процесс является асинхронным.

Как и сеансовые компоненты без состояния, компоненты, управляемые сообщениями, не поддерживают никакого состояния диалога. Обработка каждого нового сообщения происходит независимо от предыдущего. Компоненты, управляемые сообщениями, обсуждаются более подробно в главе 13.

Соглашения между компонентом и контейнером

Среда, окружающая компонент, находящийся на сервере, часто называется *контейнером* (*container*). Контейнер больше является концеп-

цией, чем физической конструкцией. По существу, контейнер действует как посредник между компонентом и сервером EJB. Он создает и управляет компонентными и внутренними объектами, соответствующими отдельному типу компонента, и помогает этим конструкциям управлять ресурсами компонентов и использовать во время выполнения основные службы, такие как транзакции, безопасность, параллелизм, систему имен и т. д. В принципе, у сервера может быть много контейнеров, каждый из которых может содержать один или несколько типов компонентов. Как вы увидите немного позже, контейнер и сервер не являются совершенно разными конструкциями, но спецификация EJB определяет компонентную модель в терминах обязанностей контейнера, поэтому здесь мы будем придерживаться этих соглашений.

Компоненты EJB взаимодействуют с контейнером EJB с помощью известной компонентной модели. Основу этой модели составляют интерфейсы `EntityBean`, `SessionBean` и `MessageDrivenBean` (в EJB 2.0). Как мы узнали чуть раньше, эти интерфейсы предоставляют методы обратного вызова, уведомляющие класс компонента о событиях жизненного цикла. Во время выполнения контейнер вызывает эти методы экземпляра компонента, когда наступает подходящее событие жизненного цикла. Например, если контейнер собирается записать состояние экземпляра объектного компонента в базу данных, он сначала вызывает метод `ejbStore()` этого экземпляра. Что дает экземпляру компонента возможность сделать некоторую «приборку» своего состояния непосредственно перед его записью в базу данных. Метод `ejbLoad()` вызывается сразу после того, как состояние компонента было заполнено из базы данных, давая возможность разработчику компонента обработать его состояние перед первым вызовом прикладного метода.[1] Другие методы обратного вызова могут использоваться классом компонента похожим образом. EJB определяет, когда вызываются эти методы и что должно быть сделано внутри их контекста. Это дает разработчику компонента предсказуемую компонентную модель времени выполнения.

Хотя методы обратного вызова объявлены в компонентных интерфейсах, их действительная реализация не обязательна. Другими словами, тело метода в классе компонента для каждого или всех методов обратного вызова может быть оставлено пустым. Компоненты, реализующие один или несколько методов обратного вызова, являются обычно более изощренными и обращаются к ресурсам, не поддерживаемым системой EJB. В эту категорию часто попадают компоненты, служащие «оберткой» для существующих систем. Единственное исключение из этого – метод `onMessage()`, определенный в интерфейсе `Message-`

[1] Описанное здесь поведение методов `ejbLoad()` и `ejbStore()` относится к постоянству, управляемому контейнером. В постоянстве, реализуемом компонентом, их поведение несколько отличается. Эти различия подробно рассматриваются в главе 11.

DrivenBean. У этого метода должна быть действительная реализация, если компоненту, управляемому сообщениями, необходимо выполнять что-нибудь полезное.

javax.ejb.EJBContext – это интерфейс, реализуемый контейнером и также являющийся частью соглашения между компонентом и контейнером. Объектные компоненты используют подкласс интерфейса javax.ejb.EJBContext с именем javax.ejb.EntityContext. Сеансовые – подкласс с именем javax.ejb.SessionContext, а компоненты, управляемые сообщениями, – подкласс javax.ejb.MessageDrivenContext. Эти типы EJBContext предоставляют компоненту информацию о его контейнере, клиенте, использующем компонент, и о самом компоненте. Также они предоставляют другие функциональные возможности, подробно описываемые в главах 11–13. Важная особенность типов EJBContext состоит в том, что они предоставляют компоненту информацию об окружающем мире, на основании которой компоненты могут обрабатывать запросы от клиентов и методов обратного вызова от контейнера.

В дополнение к EJBContext EJB 1.1 и 2.0 расширяют интерфейс между компонентом и контейнером, чтобы включить в него пространство имен JNDI, называемое *окружающим контекстом имен* (*environment naming context*). Окружающий контекст имен JNDI подробно обсуждается в главах 11 и 12.

Соглашения между контейнером и сервером

Соглашения между контейнером и сервером не определяются спецификацией Enterprise JavaBeans. Они оставлены неопределенными для того, чтобы дать производителям максимальную гибкость при определении своих серверных технологий EJB. Кроме изоляции компонентов от сервера обязанности контейнера в системе EJB несколько неясны. В спецификации EJB определены только соглашения между компонентами и контейнером, а не между контейнером и сервером. Например, трудно точно определить, где кончается контейнер и начинается сервер, когда дело доходит до управления ресурсами и другими службами.

В первых поколениях серверов EJB эта неопределенность еще не была проблемой из-за того, что большинство производителей серверов EJB также поставляли и контейнеры. Поскольку производитель предоставлял и контейнер, и сервер, интерфейс между ними мог остаться их собственный. Однако в последующих поколениях спецификации EJB может быть проведена определенная работа по определению интерфейса между контейнером и сервером, и очерчены обязанности контейнера.

Одно из преимуществ определения интерфейса между контейнером и сервером состоит в том, что он позволит третьим производителям выпускать контейнеры, способные подключаться к серверу EJB. Если обязанности контейнера и сервера буду четко определены, то произво-

дители, специализирующиеся в технологиях, поддерживающих эти разные обязанности, смогут сконцентрироваться на создании контейнеров или серверов, лучше соответствующих их основной компетенции. Недостатком четкого определения интерфейса между контейнером и сервером является то, что подход «plug-and-play» может повлиять на производительность. Высокий уровень абстракции, необходимый для четкого отделения интерфейса контейнера от сервера, естественно, приведет к ослаблению связей между этими большими компонентами, что может вызвать снижение производительности. Следующее правило лучше всего описывает преимущества и недостатки, связанные с интерфейсом между контейнером и сервером: чем теснее интеграция, тем выше производительность. Самым отпугивающим в определении интерфейса между контейнером и сервером является то, что потребуется определение низкоуровневых деталей, что было одной из проблем, поставленных технологией CORBA перед производителями CTM. Возможность реализации низкоуровневых сервисов, таких как транзакции и безопасность, является для производителей самой привлекательной особенностью EJB.[1]

Многие EJB-совместимые серверы на самом деле поддерживают несколько разных видов технологий промежуточного программного обеспечения (middleware). Например, для сервера EJB достаточно обычное дело – поддержка специфичной для производителя модели CTM, а также EJB, сервлетов, функциональности веб-сервера, драйвера JMS и других серверных технологий. Определение понятия контейнера EJB полезно для четкого отделения части сервера, поддерживающей EJB, от других служб.

Другими словами, мы могли бы определить обязанности контейнера и сервера, опираясь на текущую реализацию спецификации EJB. Иначе говоря, мы могли бы изучить, как производители определяют контейнер в своих серверах, и использовать его в качестве руководства. К несчастью, обязанности контейнера в каждом сервере EJB сильно зависят от основной компетенции производителя в этом вопросе. Например, производители баз данных реализуют контейнеры не так, как производители TP-мониторов. Стратегии назначения обязанностей контейнера и сервера так сильно варьируются, что раздельное их обсуждение не поможет вам по-настоящему понять общую архитектуру. Вместо этого данная книга рассматривает архитектуру системы EJB так, как будто контейнер и сервер являются одним компонентом, и называет его собирательно сервером или контейнером EJB.

---

[1] Из всех доступных сегодня серверов EJB, как коммерческих, так и с открытым кодом, только в одном (OpenEJB) проводились эксперименты по определению интерфейса между контейнером и сервером. OpenEJB – это контейнерная система EJB с открытым исходным кодом, разрабатываемая мной.

Выводы

В этой главе были рассмотрены основы архитектуры системы EJB. К настоящему моменту вы должны понимать, что компоненты – это прикладные объекты. Внутренние интерфейсы определяют методы жизненного цикла для создания, поиска и ликвидации компонента, а удаленные и локальные интерфейсы определяют открытые прикладные методы компонента. У компонентов, управляемых сообщениями, нет компонентных интерфейсов. Именно в классе компонента реализуются состояние и поведение компонента.

Существуют три основных вида компонента: объектные, сеансовые и компоненты, управляемые сообщениями. Объектные компоненты являются постоянными и представляют лицо, место или предмет. Сеансовые компоненты расширяют клиентское приложение и воплощают процесс или рабочий поток, определяющий взаимодействие других компонентов. Сеансовые компоненты – непостоянные, принимают свое состояние от клиента и существуют только до тех пор, пока они нужны клиентскому приложению. Компоненты, управляемые сообщениями, в EJB 2.0 представляют собой точки подключения, позволяющие другим приложениям взаимодействовать с приложениями EJB при помощи системы асинхронных сообщений JMS. Компоненты, управляемые сообщениями, как и сеансовые компоненты, не постоянны и не поддерживают состояние диалога.

Компонентный и внутренний объекты являются концептуальными конструкциями, перенаправляющими вызовы методов от клиента сеансовым и объектным компонентам и помогающими контейнеру управлять компонентами во время выполнения. Клиенты объектных и сеансовых компонентов не обращаются напрямую к экземплярам класса компонента. Вместо этого клиентское приложение взаимодействует с заглушками, соединенными с компонентным и внутренним объектами. Компонентный объект реализует удаленный и/или локальный интерфейс и расширяет возможности класса компонента. Внутренний объект реализует внутренний интерфейс и тесно взаимодействует с контейнером с целью создания, поиска и удаления компонентов.

Компоненты взаимодействуют со своими контейнерами, следуя стандартным соглашениям. Эти соглашения включают в себя методы обратного вызова, объект `EJBContext` и контекст имен окружения JNDI. Методы обратного вызова уведомляют класс компонента о наступлении событий его жизненного цикла. `EJBContext` и контекст имен окружения JNDI снабжают экземпляр компонента информацией о его окружении. Соглашения между контейнером и сервером не стандартизованы и на данный момент остаются частными. В будущих версиях спецификации эти соглашения могут быть определены.

3

Управление ресурсами и основные службы

В главе 2 были рассмотрены основы архитектуры Enterprise JavaBeans, включая отношения между классом компонента, компонентными интерфейсами, компонентным объектом, внутренним объектом и контейнером EJB. Эти архитектурные компоненты определяют общую модель распределенных серверных компонентов и мониторов компонентных транзакций.

Одна из причин, благодаря которым СТМ представляют собой такие мощные платформы распределенных объектов, состоит в том, что они занимаются не только распределением объектов: они также управляют ресурсами, используемыми распределенными объектами. СТМ разработаны для управления одновременно тысячами и даже миллионами распределенных объектов. Для того чтобы при этом устойчиво работать, СТМ должны быть очень интеллектуальными менеджерами ресурсов, управляя тем, как распределенные объекты используют память, потоки, подключения к базам данных, мощность процессора и т. д. Спецификация EJB указывает на то, что некоторые из методов управления ресурсами, используемых СТМ, широко распространены, и поэтому она определяет интерфейсы, помогающие разработчикам создавать компоненты, которые могут пользоваться этими привычными средствами.

EJB СТМ также являются мощными посредниками для распределенных объектов. Мало того, что они помогают клиентам находить тре-

буемые им распределенные объекты, но они также предоставляют множество услуг, которые значительно упрощают правильное использование объектов клиентом. СТМ обычно поддерживают шесть основных служб: параллелизм, управление транзакциями, постоянство, распределение объектов, систему имен и безопасность. Эти услуги предоставляют определенного рода инфраструктуру, необходимую для построения удачной трехуровневой системы.

С введением в EJB 2.0 компонентов, управляемых сообщениями, Enterprise JavaBeans выходит за рамки обычных СТМ, расширяя обязанности платформы, включающие теперь еще и управление компонентами асинхронных сообщений. Исторически сложилось так, что СТМ отвечают только за управление распределенными объектами на базе RMI. Несмотря на то что для компонентов, управляемых сообщениями, метод доступа отличается, EJB все еще отвечает за управление основными службами для компонентов, управляемых сообщениями, точно так же, как и для сеансовых и объектных компонентов.

В этой главе рассматриваются как средства управления ресурсами, так и основные службы, доступные в Enterprise JavaBeans.

Управление ресурсами

Одно из основных преимуществ серверов EJB состоит в том, что они сохраняют работоспособность при больших нагрузках, продолжая поддерживать высокий уровень производительности. Большая прикладная система с многими пользователями запросто может потребовать одновременного использования тысяч и даже миллионов объектов. С увеличением количества взаимодействий между этими объектами средства, связанные с обработкой параллелизма и транзакций, могут увеличить время ответа системы и огорчить этим пользователей. Серверы EJB позволяют увеличить производительность за счет синхронизации взаимодействий между объектами и совместного использования ресурсов.

Существует связь между количеством клиентов и числом распределенных объектов, требуемых для их обслуживания. По мере увеличения количества клиентов требуется увеличение количества распределенных объектов и ресурсов. На определенном этапе увеличение числа клиентов начнет влиять на производительность и уменьшать пропускную способность системы. EJB явно поддерживает два механизма, облегчающих управление большим количеством компонентов во время выполнения: пул экземпляров и активацию.

Пул экземпляров

Понятие пула ресурсов не является чем-то новым. Широко распространенная методика состоит в формировании пула подключений с база-

ми данных так, чтобы прикладные объекты системы могли одновременно обращаться к базам данных. Этот трюк уменьшает количество требуемых соединений с базами данных, что сокращает потребление ресурсов и увеличивает производительность. Объединение в пул и многократное использование соединений с базами данных менее накладно, чем создание и удаление соединений по необходимости. Большинство контейнеров EJB применяют объединение ресурсов в пул также и к серверным компонентам. Эта техника называется *пулом экземпляров* (*instance pooling*). Она уменьшает количество экземпляров компонентов и, следовательно, ресурсы, необходимые для обслуживания клиентских запросов. Кроме того, многократное использование экземпляров из пула не так дорого, как частое создание и уничтожение экземпляров.

Как вы уже знаете, клиенты сеансовых и объектных компонентов взаимодействуют с этими типами компонентов через удаленные и (в EJB 2.0) локальные интерфейсы, которые реализуются компонентными объектами. Клиентские приложения никогда не обращаются напрямую к реальным сеансовым и объектным компонентам. Вместо этого они взаимодействуют с компонентными объектами, содержащими в себе экземпляры компонента. Точно так же JMS-клиенты в EJB 2.0 никогда не взаимодействуют непосредственно с компонентами, управляемыми сообщениями. Они посылают сообщения, которые направляются контейнерной системе EJB. Затем контейнер EJB передает эти сообщения нужному экземпляру, управляемому сообщениями.

Применение пула экземпляров для обеспечения лучшей производительности разрешает косвенный доступ к компонентам. Другими словами, поскольку клиенты никогда не обращаются к компонентам непосредственно, нет никаких причин содержать для каждого клиента отдельную копию каждого компонента. Для выполнения своей задачи сервер может хранить гораздо меньшее количество компонентов, многократно используя каждый экземпляр для обслуживания разных запросов. Это походит на «утечку ресурсов» (resource drain), но при правильном применении сильно уменьшает количество ресурсов, реально требуемых для обслуживания всех клиентских запросов.

Жизненный цикл объектного компонента

Для того чтобы понять, как работает пул экземпляров для RMI-компонентов (сеансовых и объектных компонентов), изучим жизненный цикл объектного компонента. EJB определяет жизненный цикл объектного компонента в терминах его отношений с пулом экземпляров. Объектные компоненты могут находиться в одном из трех состояний:

Без состояния

Об экземпляре компонента говорят, что он находится в этом состоянии, если он еще не был инициализирован. Мы выделяем это сос-

тояние для того, чтобы дать жизненному циклу экземпляра компонента начало и конец.

Нахождение в пуле

Экземпляр, характеризующийся этим состоянием, был создан контейнером, но еще не был связан с компонентным объектом.

Состояние готовности

Экземпляр компонента, находящийся в этом состоянии, был связан с компонентным объектом и готов отвечать на вызовы прикладных методов.

Обзор переходов между состояниями

Каждый производитель EJB реализует пул экземпляров для объектных компонентов по-разному, но все стратегии стараются управлять коллекциями экземпляров компонентов так, чтобы они были быстро доступными во время выполнения. Чтобы создать пул экземпляров, контейнер EJB создает несколько экземпляров классов компонента и затем удерживает их, пока в них не возникнет необходимость. По мере того как клиенты выполняют запросы прикладных методов, экземплярам компонентов, находящимся в пуле, назначаются компонентные объекты, связанные с клиентами. Если компонентный объект больше не нуждается в экземпляре, последний возвращается в пул экземпляров. Сервер EJB поддерживает пулы экземпляров для каждого типа установленного компонента. Все экземпляры, находящиеся в пуле, *эквивалентны* – все они интерпретируются одинаково. Экземпляры из пула экземпляров извлекаются произвольно и при необходимости связываются с компонентными объектами.

Вскоре после того как экземпляр компонента проинициализирован и помещен в пул, ему передается ссылка на объект `javax.ejb.EJBContext`, поддерживаемый контейнером. `EJBContext` предоставляет интерфейс, обеспечивающий взаимодействие компонента с EJB-окружением. Этот `EJBContext` становится более полезным, когда экземпляр компонента перемещается в состояние готовности. У компонентов есть также и контекст JNDI, называемый контекстом имен окружения (Environment Naming Context, ENC). Функция этого окружения не существенна для данного обсуждения и будет описана более подробно далее в этой главе.

Если клиент использует внутренний объект для получения удаленной или локальной ссылки на компонент, контейнер реагирует на это созданием компонентного объекта. После создания компонентный объект связывается с экземпляром компонента, находящимся в пуле экземпляров. Когда экземпляр компонента связывается с компонентным объектом, он формально переходит в состояние готовности. В состоянии готовности экземпляр компонента может принимать за-

просы от клиента и обратные вызовы от контейнера. На рис. 3.1 показана последовательность событий, выполняемая, когда компонентный объект заключает в себя экземпляр компонента и обслуживает клиента.

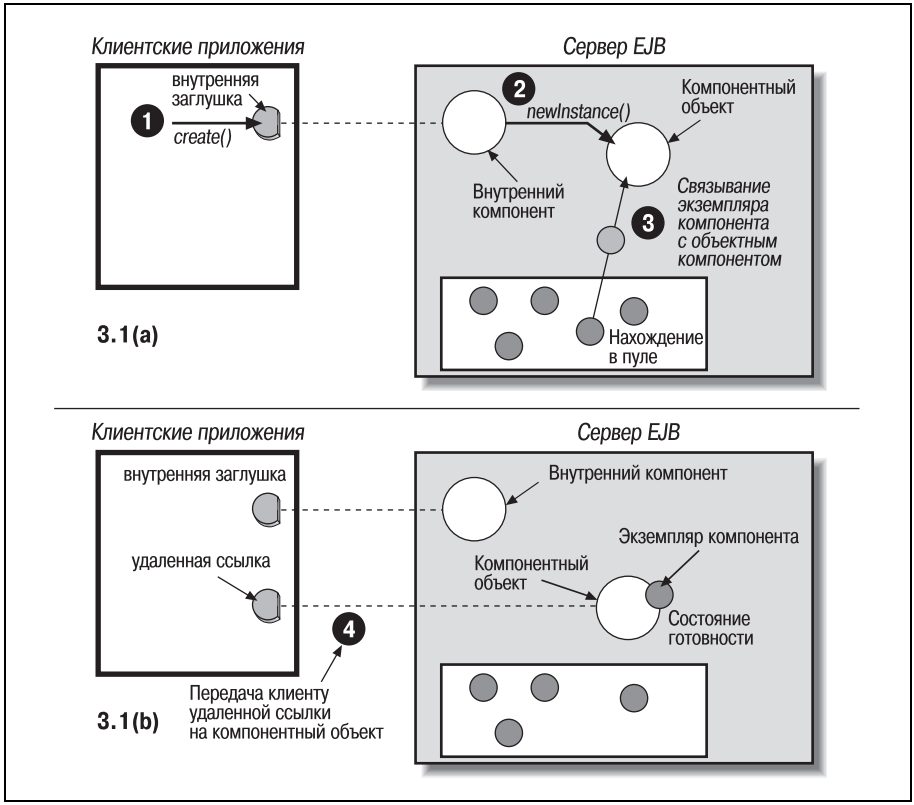

Рис. 3.1. Компонент перемещается из пула экземпляров в состояние готовности

Когда экземпляр компонента перемещается в состояние готовности, `EJBContext` берет на себя другую роль. `EJBContext` предоставляет информацию о клиенте, использующем компонент. Он также дает экземпляру доступ к своим собственным внутреннему и компонентному объектам, что полезно, если компоненту требуется передавать ссылки на самого себя другим компонентам или создать, найти или удалить компоненты своего собственного класса. Поэтому `EJBContext` не является статическим классом. Это интерфейс с контейнером, и его состояние изменяется по мере того, как экземпляр назначается разным компонентным объектам.

Если клиент завершает работу с удаленной ссылкой компонента, то либо удаленная ссылка выходит из области видимости, либо вызыва-

ется один из удаляющих методов компонента.[1] Если компонент удален или он больше не находится в области видимости, экземпляр компонента отсоединяется от компонентного объекта и возвращается в пул экземпляров. Экземпляры компонентов также могут быть возвращены в пул экземпляров во время перерыва между клиентскими запросами. Если принимается клиентский запрос, а с компонентным объектом не связан ни один экземпляр компонента, экземпляр извлекается из пула и связывается с компонентным объектом. Этот процесс называется *подкачкой экземпляров* (*instance swapping*).

Экземпляр компонента после возвращения в пул экземпляров снова готов к обслуживанию клиентских запросов. На рис 3.2 показан жизненный цикл экземпляра компонента.

Рис. 3.2. Жизненный цикл экземпляра компонента

Количество экземпляров в пуле колеблется в процессе связывания экземпляров с компонентными объектами и возвращения их в пул. Кроме этого, контейнер может управлять количеством экземпляров, находящихся в пуле, увеличивая его при возрастании активности клиентов и уменьшая на время менее активных периодов.

Подкачка экземпляров

Сеансовые компоненты без состояния особенно удобны для использования пула экземпляров. Сеансовый компонент без состояния не под-

[1] Методы, которые могут применяться для удаления компонента, определены во всех интерфейсах `EJBHome`, `EJBLocalHome`, `EJBObject` и `EJBLocalObject`.

держивает никакого состояния в промежутках между вызовами методов. Каждый вызов метода сеансового компонента без состояния работает независимо, не рассчитывая при выполнении своей задачи на переменные экземпляра. Это означает, что любой сеансовый экземпляр без состояния может обслуживать запросы любого компонентного объекта подходящего типа, позволяя контейнеру подкачивать экземпляры компонентов в том и другом направлении в промежутках между вызовами методов, выполняемых клиентом.

На рис. 3.3 показан данный тип подкачки экземпляров между вызовами методов. На рис. 3.3a экземпляр A обслуживает вызов прикладного метода, перенаправленный компонентным объектом 1. Экземпляр A, обслужив запрос, перемещается обратно в пул экземпляров (рис. 3.3b). Когда компонентный объект 2 принимает вызов прикладного метода, экземпляр A связывается с этим компонентным объектом на время выполнения операции (рис. 3.3c). Пока экземпляр A обслуживает компонентный объект 2, компонентный объект 1 принимает от клиента еще один вызов метода, который обслуживается экземпляром B (рис. 3.3d).

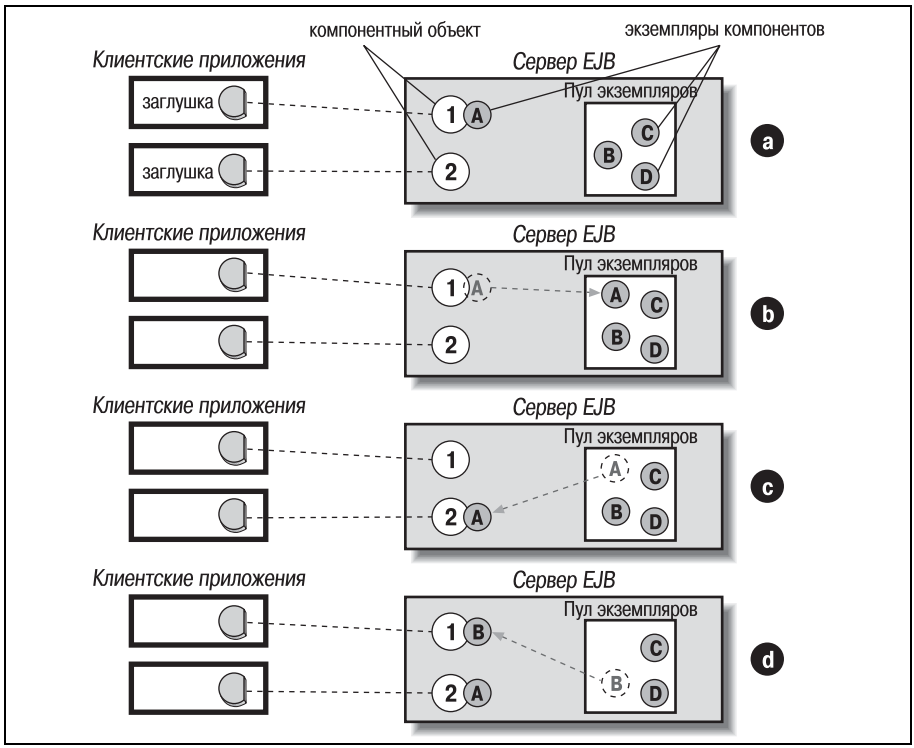

Рис. 3.3. Сеансовые компоненты без состояния и стратегия подкачки

Такая стратегия подкачки позволяет нескольким экземплярам сеансовых компонентов без состояния обслуживать сотни клиентов. Это ста-

новится возможным благодаря тому, что количество времени, требуемого на выполнение большинства вызовов методов, существенно короче пауз между их вызовами. Периоды в жизни экземпляра компонента, когда он не обслуживает активно компонентный объект, являются непродуктивными. Использование пула экземпляров минимизирует эти периоды неактивности. Экземпляр компонента, закончив обслуживание запроса компонентного объекта, сразу же становится доступным любому другому компонентному объекту, которому он может потребоваться. Это позволяет меньшему количеству сеансовых экземпляров без состояния обслуживать большее количество запросов, что уменьшает потребление ресурсов и увеличивает производительность.

Сеансовые компоненты без состояния объявляются не имеющими состояния в дескрипторе развертывания. Ничто в определении класса сеансового компонента не указывает на то, что он не имеет состояния. Если класс компонента развернут как не имеющий состояния, контейнер предполагает, что в промежутках между вызовами методов не поддерживается никакое состояние диалога. Так что компонент без состояния может иметь переменные экземпляра, но из-за того что экземпляры компонента могут обслуживать несколько разных компонентных объектов, они не должны использоваться для поддержки состояния диалога.

Реализации пула экземпляра варьируются от производителя к производителю. Один из аспектов, которым реализации пула экземпляра часто отличаются, состоит в том, как экземпляры извлекаются из пула. Двумя наиболее распространенными стратегиями являются FIFO (первым пришел – первым ушел) и LIFO (последним пришел – первым ушел). При использовании FIFO экземпляры помещаются в очередь, в которой они ожидают обслуживания компонентных объектов. LIFO использует стратегию, при которой последний компонент, добавленный к стеку, является первым компонентом, который будет связан со следующим компонентным объектом. На рис. 3.3 показана стратегия LIFO.

EJB 2.0: Компоненты, управляемые сообщениями, и пул экземпляров

Как и сеансовые компоненты без состояния, компоненты, управляемые сообщениями, не поддерживают состояние, специфичное для клиентского запроса, что делает их превосходными кандидатами на хранение в пуле.

В большинстве контейнеров EJB каждый тип компонента, управляемого сообщениями, имеет свой собственный пул экземпляров, предназначенный для обслуживания входящих сообщений. Компоненты, управляемые сообщениями, подписываются на определенные пункты назначения сообщений, которые являются своего рода адресом, используемым при посылке и приеме сообщений. Когда JMS-клиент по-

сылает асинхронное сообщение в заданный пункт назначения, сообщение пересылается контейнеру компонентов, управляемых сообщениями, которые подписались на этот пункт назначения. Сначала контейнер EJB определяет, какой компонент подписался на этот пункт назначения, затем извлекает экземпляр этого типа из пула экземпляров для обработки сообщения. После того как экземпляр компонента, управляемого сообщениями, закончит обработку сообщения (после возвращения из метода `onMessage()`), контейнер EJB вернет экземпляр в его пул экземпляров. Используя пулы экземпляров, контейнер EJB может обрабатывать сотни, возможно, тысячи сообщений одновременно. На рис. 3.4 показано, как клиентские запросы обрабатываются контейнером EJB.

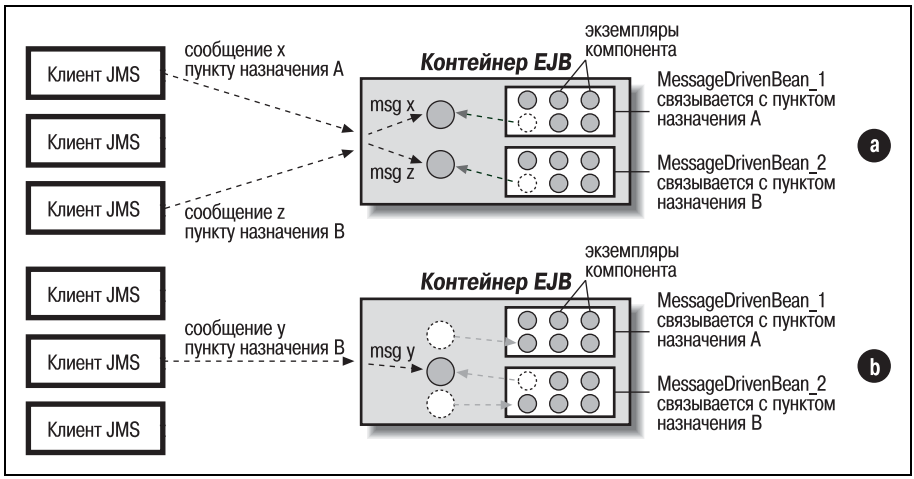

Рис. 3.4. Пул экземпляров компонента, управляемого сообщениями

Верхний JMS-клиент передает сообщение пункту назначения A, а нижний JMS-клиент передает сообщение пункту назначения B. Контейнер EJB извлекает экземпляр MessageDrivenBean_1 для обработки сообщения, предназначенного для пункта назначения A, а экземпляр MessageDrivenBean_2 – для обработки сообщения, предназначенного для пункта назначения B. Экземпляры компонентов удаляются из пула, связываются и используются для обработки этих сообщений.

Чуть позже средний JMS-клиент посылает сообщение пункту назначения B. К этому моменту первые два сообщения уже обработаны, и контейнер возвращает экземпляры в соответствующие им пулы. Поскольку поступило новое сообщение, контейнер извлекает новый экземпляр MessageDrivenBean_2 для обработки этого сообщения.

Компоненты, управляемые сообщениями, всегда развертываются для обработки сообщения из определенного пункта назначения. В вышеприведенном примере экземпляры MessageDrivenBean_1 обрабатыва-

ют сообщения только из пункта назначения A, а экземпляры MessageDrivenBean_2 обрабатывают сообщения только из пункта назначения B. Несколько сообщений из одного пункта назначения могут обрабатываться одновременно. Если, например, в пункт назначения A от сотни разных JMS-клиентов одновременно прибудет сотня сообщений, контейнер EJB просто извлечет сотню экземпляров MessageDrivenBean_1 для обработки входящих сообщений, каждому сообщению будет назначен свой экземпляр.

Способность одновременно обрабатывать сообщения делает компоненты, управляемые сообщениями, чрезвычайно мощными, помещая их в один ряд с сеансовыми и объектными компонентами. Они действительно являются первоклассными компонентами и важным расширением платформы Enterprise JavaBeans.

Механизм активации

В отличие от других компонентов, сеансовые компоненты с состоянием сохраняют свое состояние между вызовами методов. Оно было названо *состоянием диалога* (*conversational state*) из-за того, что оно представляет собой длительный сеанс связи с клиентом сеансового компонента с состоянием. Целостность этого состояния диалога необходимо поддерживать в течение всего сеанса обслуживания клиента. Сеансовые компоненты с состоянием, в отличие от компонентов без состояния, объектных компонентов и компонентов, управляемых сообщениями, не участвуют в формировании пула экземпляров. Вместо этого для сохранения ресурсов используется активация сеансовых компонентов с состоянием. Когда серверу EJB потребуется сохранить ресурсы, он может удалить сеансовые компоненты с состоянием из памяти. Это уменьшает количество экземпляров, поддерживаемых системой. Для того чтобы пассивировать («деактивировать») компонент и сохранить его состояние диалога, состояние компонента сериализуется во внешнюю память и поддерживается связанным с его компонентным объектом. Когда клиент вызывает метод компонентного объекта, происходит создание нового экземпляра с состоянием, восстановленным из внешней памяти.

Пассивация (*passivation*) – это процесс отсоединения экземпляра компонента с состоянием от его компонентного объекта и сохранения его состояния. Пассивация требует, чтобы состояние экземпляра компонента было связано с его компонентным объектом. После пассивации компонента его экземпляр можно безопасно удалить из компонентного объекта и из памяти. Клиенты даже не подозревают о существовании процесса деактивации. Запомните, что клиент использует удаленную ссылку компонента, которая реализуется компонентным объектом, и поэтому непосредственно не связана с экземпляром компонента. В результате, даже когда компонент пассивируется, соединение клиента с компонентным объектом может оставаться активным.

Активация (*activation*) компонента – это процесс восстановления состояния экземпляра компонента, связанного с его компонентным объектом. Когда вызывается метод пассивированного компонентного объекта, контейнер автоматически создает новый экземпляр и устанавливает его поля в значения, сохраненные во время пассивации. Компонентный объект может затем, как обычно, перенаправить вызов метода компоненту. На рис. 3.5 показаны активация и пассивация компонента с состоянием. На рис. 3.5(a) выполняется пассивация компонента. Состояние экземпляра B считывается и поддерживается относительно компонентного объекта, который он обслуживал. На рис. 3.5(b) компонент пассивирован, а его состояние сохранено. В этом месте компонентный объект не связан ни с каким экземпляром компонента. На рис. 3.5(c) выполняется активация компонента. Создается новый экземпляр, экземпляр C, связывается с компонентным объектом и заполняется сохраненным состоянием, связанным с компонентным объектом.

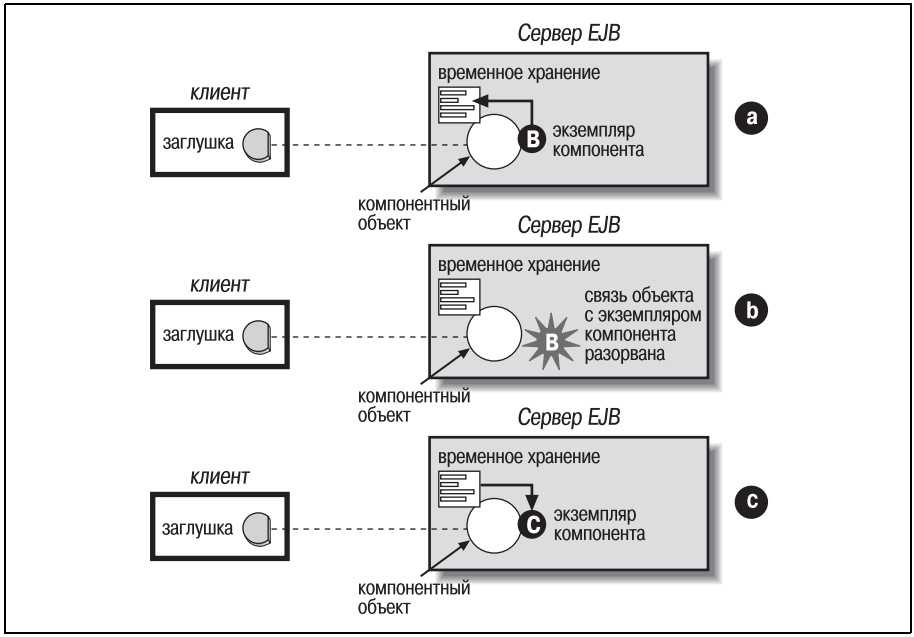

Рис. 3.5. Процессы пассивации и активации

Реализация конкретного механизма активации и пассивации компонентов с состоянием оставлена на усмотрение производителя, но каждый компонент с состоянием является сериализуемым и, таким образом, предоставляет по крайней мере один способ временного сохранения своего состояния. Хотя некоторые производители пользуются механизмом сериализации Java, конкретный механизм сохранения со-

стояния диалога не определен. Допустим любой из механизмов, в котором действуют правила временной свертки объектов, аналогичные используемым при сериализации в Java. Поскольку Enterprise JavaBeans поддерживает также и другие способы сохранения состояния компонента, свойство изменяемости при активации пассивированного компонента трактуется не так же, как в сериализации Java. В сериализации Java изменяемые (transient) поля при десериализации объекта всегда возвращаются к начальным значениям в соответствии с их типом. Целые типы устанавливаются в ноль, тип boolean – в `false`, объектные ссылки – в `null` и т. д. В EJB изменяемые поля после активации не обязательно возвращаются к своим начальным значениям, а могут поддерживать свои оригинальные значения или любое произвольное значение. Работа с изменяемыми полями требует осторожности, т. к. их состояние после активации зависит от конкретной реализации.

Процесс активации обеспечивается методами обратного вызова, управляющими жизненным циклом, рассмотренными в главе 2. Методы `ejbActivate()` и `ejbPassivate()`, соответственно уведомляют экземпляр компонента с состоянием о том, что его собираются активировать или пассивировать. Метод `ejbActivate()` вызывается сразу же после успешной активации экземпляра компонента и может применяться для установки изменяемых полей в начальные значения, если есть такая необходимость. Метод `ejbPassivate()` вызывается непосредственно перед пассивацией экземпляра компонента. Эти два метода особенно полезны в случаях, когда экземпляр компонента поддерживает соединения с ресурсами, которые должны быть обработаны или освобождены до пассивации и повторно получены при следующей активации. Поскольку экземпляр компонента с состоянием удаляется из памяти, открытые подключения к ресурсам не могут поддерживаться. Исключение составляют удаленные ссылки на другие компоненты и объект `SessionContext`, которые должны поддерживаться в сериализованном состоянии компонента и восстанавливаться при активации компонента. EJB также требует, чтобы ссылки на контекст окружения JNDI, компонентные интерфейсы и компонент UserTransaction сохранялись в процессе пассивации.

В отличие от компонентов с состоянием, объектные компоненты не содержат состояния диалога, требующего сериализации. Вместо этого состояние каждого экземпляра объектного компонента сохраняется непосредственно в базе данных. Объектные компоненты, однако, используют методы обратного вызова, управляющие активацией (`ejbActivate()` и `ejbPassivate()`), для уведомления экземпляра, когда его собираются перемещать из пула или в пул экземпляров. Метод `ejbActivate()` вызывается сразу же после того, как экземпляр компонента извлекается в компонентный объект, а метод `ejbPassivate()` вызывается непосредственно перед тем, как экземпляр переносится обратно в пул.

Базовые службы

Для распределенных приложений доступно множество дополнительных служб. OMG (организация, управляющая созданием стандартов CORBA), например, определяет 13 служб для использования в ORB, совместимых с CORBA. В этой книге рассматриваются семь дополнительных служб, называемых *базовыми службами* (*primary services*), поскольку для законченной платформы Enterprise JavaBeans они обязательны. Базовые службы включают в себя параллелизм, транзакции, постоянство, распределенные объекты, систему асинхронных сообщений (для EJB 2.0), систему имен и безопасность.

Эти семь базовых служб не являются новыми понятиями. Некоторое время назад OMG определила интерфейсы для этих служб, специфичные для платформы CORBA. В большинстве традиционных CORBA ORB службы представляют собой дополнительные подсистемы, используемые прикладным кодом явно. Это означает, что разработчик серверного компонента должен помещать код, основанный на API базовой службы, вместе со своей прикладной логикой. Применение базовых служб становится сложным, когда они сочетаются с методами управления ресурсами, поскольку базовые службы сами по себе достаточно сложны. Использование их в комбинации только усугубляет проблему.

По мере того как требуются все более сложные взаимодействия между компонентами, координация этих служб становится сложной задачей, требующей знаний системного уровня, не связанных с задачей создания прикладной логики приложения. Разработчики приложений могут так сильно завязнуть в этих системных проблемах по координации различных базовых служб и механизмов управления ресурсами, что их основная обязанность – моделирование прикладной задачи – будет вовсе забыта.

Серверы EJB выполняют автоматическое управление всеми базовыми службами. Это освобождает разработчиков приложения от изучения этих достаточно сложных служб. И они могут сосредоточиться на создании прикладной логики, описывающей систему, а заботу о системном уровне возложить на сервер EJB. В следующих разделах мы рассмотрим все базовые службы и узнаем, как они поддерживаются EJB.

Параллелизм

Проблема параллелизма важна для компонентов всех типов, но имеет различное значение, когда применяется к компонентам, управляемым сообщениями (в EJB 2.0), и сеансовым и объектным компонентам, основанным на RMI. Это происходит из-за различий в контексте: в случае с компонентами на базе RMI параллелизм связан с многочисленными клиентами, обращающимися к одному компоненту одновре-

менно, а в случае с компонентами, управляемыми сообщениями, параллелизм связан с одновременной обработкой многочисленных асинхронных сообщений. По этой причине мы рассмотрим важность параллелизма как базовой службы отдельно для разных типов компонентов.

Параллелизм в сеансовых и объектных компонентах

Сеансовые компоненты не поддерживают параллельный доступ. И это имеет смысл, что становится понятным, если рассмотреть характер сеансовых компонентов с состоянием и без него. Компонент с состоянием представляет собой расширение одного клиента и обслуживает только его. Не имеет смысла делать компоненты с состоянием параллельными, если они используются только клиентами, которые их создали. Сеансовым компонентам без состояния нет необходимости быть параллельными, поскольку они не поддерживают состояние, которое требуется использовать совместно. Область действия операций, выполняемых компонентом без состояния, ограничена областью каждого вызова метода. Состояние диалога не поддерживается.

Объектные компоненты представляют данные в базе данных, которая используется совместно и должна быть доступна параллельно. Объектные компоненты – это разделяемые компоненты. В EJB-системе «Титан», например, имеются только три судна: «Paradise», «Utopia» и «Valhalla». В любой момент к объектному компоненту Ship, представляющему судно «Utopia», может обращаться сотня клиентов. Для того чтобы сделать возможным параллельный доступ к объектным компонентам, контейнер EJB должен защищать данные, представляемые разделяемым компонентом, и в то же время разрешать многочисленным клиентам одновременно обращаться к компоненту.

В системе распределенных объектов проблемы возникают тогда, когда мы пытаемся разрешить совместное использование распределенных объектов несколькими клиентами. Если два клиента работают с одним и тем же компонентным объектом, то как защитить изменения, сделанные одним клиентом, от перезаписи другим? Если, например, один клиент читает состояние экземпляра непосредственно перед тем, как другой клиент вносит изменения в тот же самый экземпляр, то данные, которые прочитал первый клиент, становятся недействительными. На рис. 3.6 показаны два клиента, совместно использующие один и тот же компонентный объект.

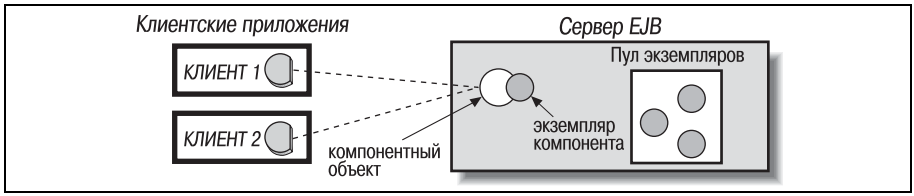

Рис. 3.6. Клиенты, совместно обращающиеся к компонентному объекту

EJB предупреждает опасности, связанные с параллелизмом в объектных компонентах, реализуя простое решение: по умолчанию EJB запрещает параллельный доступ к экземплярам компонента. Другими словами, с одним компонентным объектом могут быть связаны несколько клиентов, но к экземпляру компонента в одно и то же время может обращаться только один поток клиента. Если, например, один из клиентов вызывает метод компонентного объекта, никакой другой клиент не может обращаться к тому же экземпляру компонента, пока не завершится вызов метода. Фактически, если метод представляет собой часть большой транзакции, то до ее завершения к экземпляру компонента нельзя будет обратиться вообще, кроме как изнутри этого транзакционного контекста.

Поскольку серверы EJB обрабатывают параллелизм автоматически, нет нужды делать методы компонента потокобезопасными. Фактически спецификация EJB запрещает применение ключевого слова `synchronized`. Запрещение использования примитивов синхронизации потоков избавляет разработчиков от размышлений над тем, как им управлять синхронизацией и улучшать производительность экземпляров компонентов во время выполнения. Кроме этого, EJB явно запрещает компонентам создавать свои собственные потоки. Другими словами, как разработчик компонента, вы не можете создать поток внутри компонента. Контейнер EJB должен иметь полный контроль над компонентом для того, чтобы правильно управлять параллелизмом, транзакциями и постоянством. Разрешение разработчику компонента произвольно создавать потоки поставило бы под угрозу способность контейнера следить за тем, чем занимается компонент, и, таким образом, не позволило бы контейнеру управлять базовыми службами.

Повторная входимость. Говоря о параллелизме в объектных компонентах, мы должны рассмотреть связанное с ним понятие *повторной входимости* (*reentrance*). Повторная входимость возникает тогда, когда поток выполнения пытается повторно войти в экземпляр компонента. В EJB экземпляры объектных компонентов по умолчанию нереентерабельны, это означает, что циклические вхождения (loopbacks) не разрешаются. Прежде чем я объясняю, что такое циклические вхождения, важно, чтобы вы поняли очень важный принцип EJB: объектные и сеансовые компоненты взаимодействуют с помощью удаленных ссылок друг на друга, а не взаимодействуют непосредственно. Другими словами, когда компонент A оперирует компонентом B, он делает это таким же способом, что и прикладной клиент, используя удаленный или локальный интерфейс компонента B, реализуемый компонентным объектом. Это позволяет контейнеру EJB вклиниваться между вызовами методов от одного компонента к другому и применять службы транзакции и безопасности.

Хотя большинство взаимодействий между компонентами в EJB 2.0 происходит с помощью локальных интерфейсов совмещенных компонентов, иногда компоненты могут взаимодействовать при помощи уда-

ленных интерфейсов. Удаленные интерфейсы обеспечивают полную прозрачность расположения. Когда взаимодействия между компонентами происходят при помощи удаленных ссылок, компоненты могут быть перемещены, возможно, на другой сервер, с минимальным влиянием на остальную часть приложения.

Независимо от того, какие используются интерфейсы – удаленные или локальные, с точки зрения компонента, обслуживающего запрос, все клиенты создаются одинаковыми. На рис. 3.7 показано, что, с точки зрения компонента, вызовы прикладных методов выполняют только клиенты. Когда вызывается прикладной метод экземпляра компонента, он не может отличить удаленный клиент-приложение от клиента-компонента.

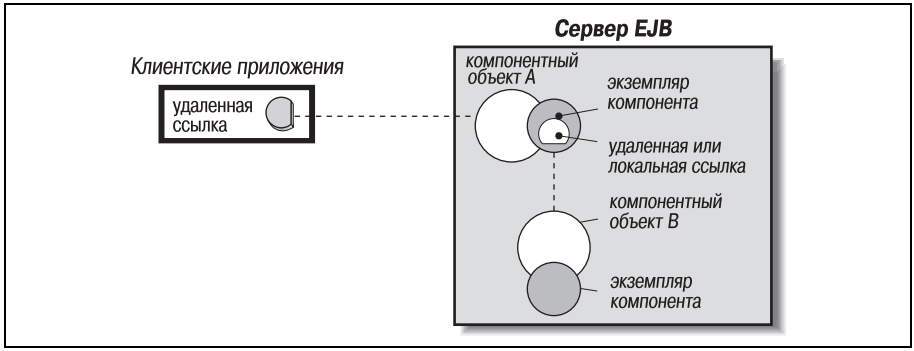

Рис. 3.7. Компоненты обращаются друг к другу через компонентные объекты

Циклическое вхождение (*loopback*) имеет место тогда, когда компонент A вызывает метод компонента B, который затем пытается выполнить обратный вызов компонента A. Этот вид взаимодействия показан на рис. 3.8. Клиент 1 вызывает метод компонента A. В ответ на этот вызов метода компонент A вызывает метод компонента B. Тут нет никакой проблемы, т. к. клиент 1 управляет доступом к компоненту A, а компонент A является клиентом компонента B. Однако если компонент B попытается вызвать метод компонента A, он будет заблокирован, поскольку поток выполнения уже вошел в компонент A. Вызывая свой вызывающий объект, компонент B выполняет циклическое вхождение. По умолчанию это не допускается, поскольку EJB не позволяет потоку выполнения повторно входить в экземпляр компонента.

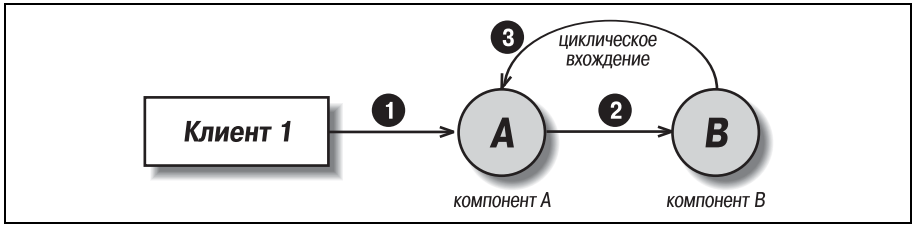

Рис. 3.8. Сценарий с циклическим вхождением

Политика нереентерабельности (запрета повторной входимости) к сеансовым и объектным компонентам применяется по-разному. Сеансовые компоненты никогда не могут быть повторно вызваны, и при попытке циклического вхождения они генерируют исключение. То же самое верно и для нереентерабельного объектного компонента. Во время развертывания объектные компоненты могут быть сконфигурированы в дескрипторе развертывания так, чтобы разрешать повторную входимость. Однако создание реентерабельного объектного компонента не рекомендуется спецификацией. Повторная входимость не применима к компонентам, управляемым сообщениями, в EJB 2.0, поскольку они не отвечают на вызовы RMI, как это делают сеансовые и объектные компоненты.

Как было рассмотрено выше, клиентский доступ к компоненту синхронизируется так, что в одно время только один клиент может обращаться к любому данному компоненту. Повторная входимость связана с потоком выполнения, созданным клиентским запросом, который пытается повторно обратиться к экземпляру компонента. Проблема с повторно входимым кодом состоит в том, что компонентный объект, который перехватывает и перенаправляет вызовы метода экземпляру компонента, не может отличить повторно входящий код от многопоточного доступа внутри одного контекста транзакции. (О контексте транзакции подробнее рассказано в главе 14.) Разрешая повторную входимость, мы также разрешаем и многопоточный доступ к экземпляру компонента. Многопоточный доступ к экземпляру компонента может привести к повреждению данных, т. к. потоки влияют на работу друг друга в то время, когда они пытаются выполнять свои собственные задачи.

Важно запомнить, что повторно входящий код отличается от экземпляра компонента, который просто вызывает свои собственные методы на уровне экземпляра. Другими словами, метод `foo()` экземпляра компонента может непосредственно вызывать свои собственные открытые, защищенные, закрытые методы и методы по умолчанию столько, сколько ему потребуется. Далее приводится пример внутреннего вызова метода экземпляра, который является совершенно законным:

```
public HypotheticalBean extends EntityBean {
    public int x;

    public double foo() {
        int i = this.getX();
        return this.boo(i);
    }
    public int getX() {
        return x;
    }
    private double boo(int i) {
        double value = i * Math.PI;
```

```
            return value;
        }
    }
```

В этом фрагменте кода прикладной метод `foo()` вызывает другой прикладной метод `getX()`, а затем закрытый метод `boo()`. Вызовы методов, выполненные изнутри тела метода `foo()`, являются внутренними вызовами экземпляра и не считаются повторными вхождениями.

EJB 2.0: Параллелизм в компонентах, управляемых сообщениями

Параллелизм в компонентах, управляемых сообщениями, относится к одновременной обработке более чем одного сообщения. Как упоминалось выше, параллельная обработка сообщений превращает компоненты, управляемые сообщениями, в мощную асинхронную компонентную модель. Если бы компоненты, управляемые сообщениями, могли обрабатывать одновременно только одно сообщение, они были бы фактически бесполезны в реальном приложении, т. к. не смогли бы обрабатывать интенсивный поток сообщений.

Одновременно в один пункт назначения могут посылать сообщения многие JMS-клиенты. Способность компонента, управляемого сообщениями, обработать все эти сообщения одновременно и называется параллелизмом. Как показано на рис. 3.9, если в одно и то же время от трех разных клиентов определенному пункту назначения посылаются три сообщения, для одновременной обработки этих сообщений будут использованы три экземпляра одного компонента, управляемого сообщениями, который подписан на этот пункт назначения или прослушивает его.

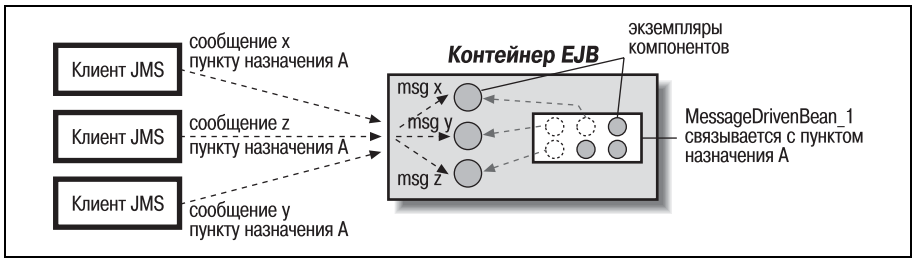

Рис. 3.9. Параллельная обработка в компонентах, управляемых сообщениями

В действительности в компонентах, управляемых сообщениями, выполняется намного больше параллельной обработки. Например, пункты назначения типа темы и очереди обрабатываются по-разному, хотя основное значение параллельной обработки остается таким же. Подробности, относящиеся к пунктам назначения типа темы и очереди, будут рассмотрены в главе 13.

Транзакции

Мониторы компонентных транзакций были созданы для того, чтобы добавить устойчивую, масштабируемую транзакционную целостность традиционных мониторов TP к динамичному миру распределенных объектов. Enterprise JavaBeans как серверная компонентная модель для CTM предоставляет мощную поддержку транзакций для всех типов компонентов (сеансовых, объектных и управляемых сообщениями).

Транзакция (*transaction*) – это единица работы, или набор задач, выполняемых вместе. Транзакции являются атомарными. Другими словами, для того чтобы транзакция считалась завершенной успешно, должны быть успешно завершены все задачи, составляющие транзакцию. В предыдущей главе для описания того, как сеансовый компонент управляет взаимодействиями между другими компонентами, мы использовали компонент TravelAgent. Здесь приведен фрагмент кода, иллюстрирующий метод `bookPassage()`, описанный в главе 2:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
    if (customer == null ||cruise == null ||cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome = (ReservationHomeRemote)
            getHome("ReservationHome",ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer,cruise,cabin,price,new Date());
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            getHome("ProcessPaymentHome",ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer,card,price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

Метод `bookPassage()` состоит из двух задач, которые должны быть закончены вместе: создание нового компонента Reservation и обработка оплаты. Когда компонент TravelAgent используется для регистрации пассажира, должны быть успешно выполнены и прием денег с кредитной карточки пассажира, и выполнение резервирования. Было бы недопустимо, если бы компонент ProcessPayment принял платеж с кредитной карточки, когда создание нового компонента Reservation закончилось неудачей. Точно так же нельзя выполнить резервирование,

если с кредитной карточки пассажира невозможно принять оплату. Сервер EJB контролирует транзакцию для того, чтобы гарантировать, что все задачи завершены успешно.

Управление транзакциями выполняется автоматически, поэтому разработчик компонента не должен использовать никакой API для явного управления транзакцией компонента. Просто объявление атрибутов транзакции во время развертывания укажет серверу EJB, как требуется управлять компонентом во время выполнения. EJB предоставляет механизм, позволяющий компонентам при необходимости явно управлять транзакциями. Установка атрибутов транзакции во время развертывания рассмотрена в главе 14, как и явное управление транзакциями и другие вопросы, относящиеся к транзакциям.

Постоянство

Объектные компоненты представляют поведение и данные, связанные с реальными людьми, местами или предметами. В отличие от сеансовых компонентов и компонентов, управляемых сообщениями, объектные компоненты являются постоянными. Это означает, что состояние объекта постоянно хранится в базе данных. Это позволяет объектам быть долговременными, т. е. их поведение и данные могут быть доступны в любое время, при этом не надо беспокоиться, что информация будет потеряна из-за системного сбоя.

Когда состояние компонента автоматически обрабатывается службой постоянства, за синхронизацию полей экземпляра объектного компонента и данных в базе данных отвечает контейнер. Это автоматическое постоянство называется *постоянством, управляемым контейнером* (*container-managed persistence*). Если компонент разрабатывается так, чтобы самостоятельно управлять своим собственным состоянием, как это обычно происходит при работе с устаревшими системами, оно называется *постоянством, управляемым компонентом* (*bean-managed persistence*).

Каждый производитель должен выбрать конкретный механизм реализации постоянства, управляемого контейнером, но его реализация должна поддержать методы обратного вызова EJB и транзакции. Наиболее общие механизмы, используемые при реализации постоянства производителями EJB, – это *постоянство типа «объект-отношение»* и *постоянство «объектной базы данных»*.

Постоянство «объект–отношение»

Постоянство «объект-отношение», возможно, является сегодня наиболее распространенным механизмом постоянства, используемым в серверах EJB. Постоянство «объект-отношение» включает в себя отображение состояния объектного компонента на таблицы и столбцы реляционной базы данных.

В системе «Титан» CabinBean моделирует прикладное понятие каюты судна. В CabinBean определено три поля: name типа String, deckLevel типа int и id типа Integer.

Для EJB 2.0 сокращенное определение CabinBean выглядит так:

```
public abstract class CabinBean implements javax.ejb.EntityBean {

    public abstract String getName();
    public abstract void setName(String str);

    public abstract int getDeckLevel();
    public abstract void setDeckLevel(int level);

    public abstract Integer getId();
    public abstract void setId(Integer id);

}
```

В EJB 2.0 абстрактные методы доступа представляют управляемые контейнером поля объектных компонентов, которые мы будем называть просто полями. При развертывании объектного компонента контейнер реализует эти виртуальные поля компонента, поэтому об абстрактных методах доступа можно думать как об описаниях постоянных полей. Например, говоря о состоянии, представленном абстрактными методами доступа setName()/getName(), мы будем ссылаться на него как на поле name. Точно так же getId()/setId() — это поле id, а getDeckLevel()/setDeckLevel() — поле deckLevel.

Для EJB 1.1 определение CabinBean выглядит так:

```
public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;

}
```

В случае с отображением баз данных типа «объект-отношение» поля объектного компонента сопоставляются со столбцами в реляционной базе данных. Поле name компонента Cabin, например, соответствует столбцу с именем NAME таблицы с именем CABIN реляционной базы данных «Титана». На рис. 3.10 показано графическое описание этого типа отображения.

По-настоящему хорошие системы EJB для сопоставления таблиц реляционной базы данных с полями классов объектного компонента предоставляют мастеры или интерфейсы администрирования. Применение этих мастеров, связывающих объекты с таблицами, представляет собою достаточно простой процесс, обычно выполняемый во время развертывания. На рис. 3.11 показан мастер отображения типа «объект-отношение» сервера приложений Pramati.

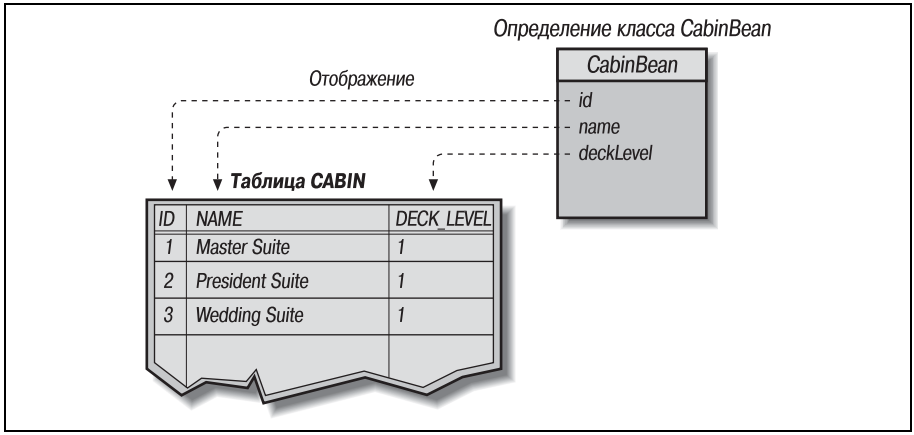

Рис. 3.10. Отображение типа «объект-отношение» объектных компонентов

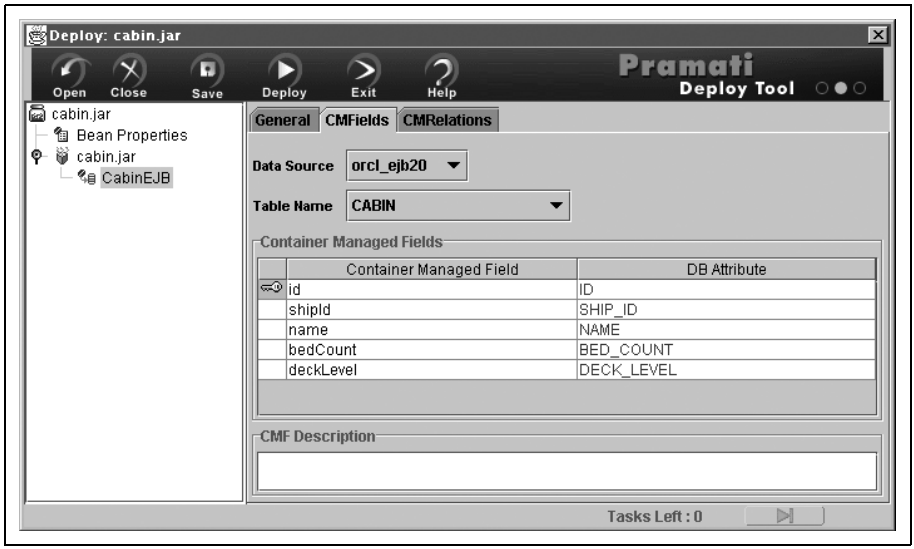

Рис. 3.11. Мастер отображения «объект-отношение» Pramati

После того как поля компонента связаны с реляционной базой данных, контейнер берет на себя ответственность за поддержание согласованности состояния экземпляра объектного компонента с соответствующими таблицами базы данных. Этот процесс называется *синхронизацией* (*synchronizing*) состояния экземпляра компонента. В случае `CabinBean` во время выполнения экземпляры компонента будут связаны отношениями «один-к-одному» со строками таблицы `CABIN` реляционной базы данных. Когда выполняется изменение компонента Cabin, он записывается в нужную строку базы данных. Достаточно часто типы компонентов будут сопоставлены больше чем с одной таблицей. Они являются более сложными типами отображения, зачастую требующи-

ми операции объединения SQL. Хорошие средства развертывания EJB должны предоставлять мастеры, которые делают создание множественных отображений достаточно легким.

Кроме этого, в постоянстве, управляемом контейнером, в EJB 2.0 определены поля отношений объектных компонентов, позволяющие этим компонентам поддерживать с другими компонентами отношения типа «один-к-одному», «один-ко-многим» и «многие-ко-многим». Объектные компоненты могут содержать коллекции других объектных компонентов или отдельные ссылки. Постоянство объектных компонентов в EJB 2.0 намного сложнее и мощнее того, которое поддерживалось в предыдущих версиях спецификации. Новая модель постоянства, управляемого контейнером, в EJB 2.0 рассматривается в главах 6, 7 и 8.

В дополнение к синхронизации состояния объекта EJB предоставляет механизмы для создания и удаления объектов. Запросы для внутреннего объекта по созданию и удалению объектов вызовут соответствующие вставки или удаление записей из базы данных. Поскольку каждый объект хранит свое состояние в таблице базы данных, новые записи (и, следовательно, уникальные компоненты) могут быть добавлены к таблицам извне системы EJB. Другими словами, вставка записи в таблицу `CABIN`, выполненная либо EJB, либо прямым доступом к базе данных, приведет к созданию нового объекта Cabin. Он создается не в смысле создания экземпляра объекта Java, а скорее в том смысле, что данные, которые описывают объект Cabin, будут добавлены к системе.

Постоянство объектной базы данных

Объектно-ориентированные базы данных предназначены для хранения объектных типов и объектных графов и поэтому намного лучше подходят для компонентов, написанных на объектно-ориентированном языке типа Java. Они предлагают более ясное отображение объектных компонентов на базу данных, чем традиционные реляционные базы данных. Однако это преимущество больше относится к EJB 1.1, чем к EJB 2.0. Постоянство, управляемое контейнером, в EJB 2.0 предоставляет программную модель, которая достаточно выразительна для того, чтобы подойти и для отображения типа «объект-отношение», и для объектных баз данных.

Хотя объектные базы данных хорошо работают, когда они имеют дело с очень сложными объектными графами, они все еще являются достаточно новыми для прикладных систем и применяются не так широко, как реляционные базы данных. В результате этого они не так хорошо стандартизированы, как реляционные базы данных, что затрудняет переход от одной базы данных к другой. Кроме этого, существует гораздо меньшее количество продуктов от третьих производителей (таких как генераторы отчетов и информационные хранилища), которые поддерживают объектные базы данных.

Некоторые реляционные базы данных поддерживают расширенные возможности для встроенного объектного постоянства. Эти базы данных позволяют некоторым объектам быть сохраненными в таблицах реляционной базы данных подобно другим типам данных и предоставляют некоторые дополнительные преимущества по сравнению с остальными базами данных.

Существующее постоянство

EJB часто используется для создания объектной оболочки для действующих систем, систем, работающих на мэйнфреймах или на основе нереляционных баз данных. Постоянство, управляемое контейнером, в таком окружении требует особого контейнера EJB, специально разработанного для доступа к данным таких систем. Производители могли бы, например, предоставлять средства отображения, позволяющие связывать компоненты с системами IMS, CICS, b-trieve и некоторыми другими существующим приложениями.

Независимо от типа используемой системы, постоянство, управляемое контейнером, предпочтительней постоянства, реализуемого компонентом. В случае постоянства, управляемого контейнером, управление состоянием компонента осуществляется автоматически, что более эффективно во время выполнения и более производительно во время создания компонента. Однако многие проекты требуют, чтобы компоненты извлекали свое состояние из существующих систем, которые не поддерживаются производителями EJB. В этих случаях разработчики должны использовать постоянство, реализуемое компонентом, а это означает, что разработчик не использует автоматическую службу постоянства сервера EJB. В главах 6–11 постоянство, управляемое контейнером и компонентом, описывается более подробно.

Распределенные объекты

Сегодня доступны три основные службы распределенных объектов: CORBA IIOP, Java RMI и Microsoft .NET. Каждая из этих платформ использует разные сетевые протоколы RMI, но все они реализуют примерно одну и ту же функцию: прозрачность расположения. Платформа Microsoft .NET, основанная на DCOM, применяется в среде Microsoft Windows и в настоящее время не поддерживается другими операционными системами. Ее тесная интеграция с продуктами Microsoft делает ее хорошим выбором для систем, построенных исключительно на платформе Microsoft. Эта ситуация может измениться с ростом поддержки простого протокола доступа к объектам (Simple Object Access Protocol, SOAP), протокола на базе XML, который быстро становится популярным и предлагает средства взаимодействия с не-Microsoft приложениями. CORBA IIOP не специфична ни для конкретной операционной системы, ни для языка и традиционно считается самой открытой службой распределенных объектов из этих трех. Она представляет собой идеальный выбор при интеграции систем, созданных с ис-

пользованием нескольких языков программирования. Java RMI является абстракцией языка Java или программной моделью для всех видов протоколов распределенных объектов. Точно так же, как JDBC API может использоваться для доступа ко всем реляционным базам данных SQL, Java RMI предназначен для применения почти со всеми протоколами распределенных объектов. На практике, Java RMI традиционно ограничивался протоколом удаленных методов Java (Java Remote Method Protocol, JRMP), известным как Java RMI поверх JRMP, который может использоваться только между приложениями Java. Недавно была разработана реализация Java RMI поверх IIOP (Java RMI-IIOP) – протокол CORBA. Java RMI-IIOP является совместимой с CORBA версией Java RMI, позволяющей разработчикам использовать простоту программной модели Java RMI и, в то же время, открывая им доступ к преимуществам платформно-независимого и не зависящего от языка CORBA-протокола IIOP.[1]

Обсуждая компонентные интерфейсы и другие интерфейсы и классы EJB, используемые на стороне клиента, мы говорим о клиентском представлении системы EJB. Понятие *клиентского представления* (*client view*) EJB не включает в себя ни компонентные объекты, ни контейнер EJB, ни подкачку экземпляров, ни какой-либо другой аспект, относящийся к конкретной реализации. Все, что может интересовать удаленный клиент, компонент определяет в своих удаленном и внутреннем интерфейсах. Все остальное для него невидимо. Поскольку клиентское представление поддерживает сервер EJB, может использоваться любой протокол распределенных объектов. EJB 2.0 требует, чтобы все серверы EJB поддерживали Java RMI-IIOP, но она не ограничивает протоколы, которые может поддерживать сервер EJB, только протоколом Java RMI-IIOP.

Независимо от применяемого протокола, сервер должен поддерживать Java-клиенты, использующие клиентский программный интерфейс Java EJB. Это означает, что данный протокол должен соответствовать программной модели Java RMI-IIOP. Возможность применения Java RMI поверх DCOM кажется несколько неправдоподобной, но Java RMI поверх SOAP возможен. На рис. 3.12 показан EJB API языка Java, поддерживаемый различными протоколами распределенных объектов.

EJB также разрешает серверам поддерживать доступ к компонентам со стороны клиентов, написанных на языках, отличных от Java. Примером этого является отображение EJB-CORBA[2], разработанное компа-

---

[1] Java RMI-IIOP может взаимодействовать с CORBA ORB, поддерживающими спецификацию CORBA 2.3.1. ORB с поддержкой более старой спецификации не могут использоваться совместно с Java RMI-IIOP из-за того, что они не реализуют часть спецификации 2.3.1, связанную с передачей объектов по значению.

[2] Enterprise JavaBeans to CORBA Mapping, версия 1.1, Санджив Кришнан, Copyright 1999, Sun Microsystems.

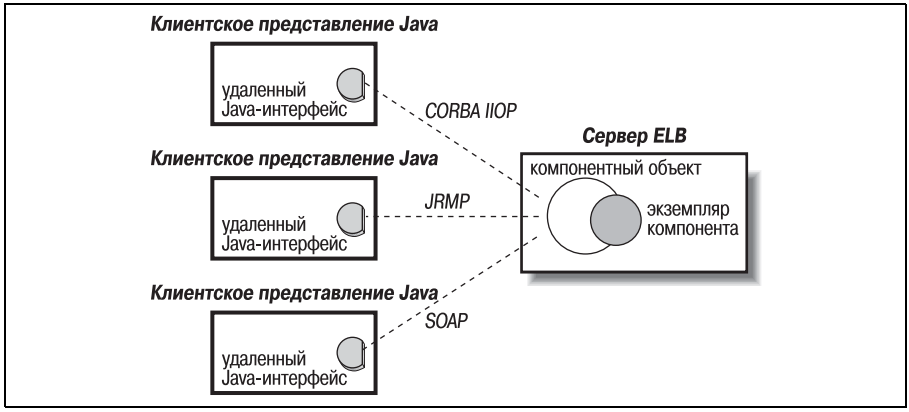

Рис. 3.12. Клиентское представление Java EJB, поддерживаемое различными протоколами

нией Sun. Этот документ описывает язык описания интерфейсов (Interface Definition Language, IDL) CORBA, который может применяться для доступа к компонентам EJB со стороны клиентов CORBA. CORBA-клиент может быть написан на любом языке, включая C++, Smalltalk, Ада и даже КОБОЛ. Отображение включает также детали, относящиеся к поддержке клиентского представления Java EJB, и детали по отображению системы имен CORBA на серверы EJB и распределенным транзакциям, включающим в себя объекты CORBA и компоненты EJB. В конце концов, может быть определено отображение EJB-SOAP, что позволит клиентским приложениям SOAP, написанным на таких языках, как Visual Basic, Delphi и PowerBuilder, обращаться к компонентам EJB. На рис. 3.13 показаны возможные способы доступа к серверу EJB со стороны разных клиентов на базе распределенных объектов.

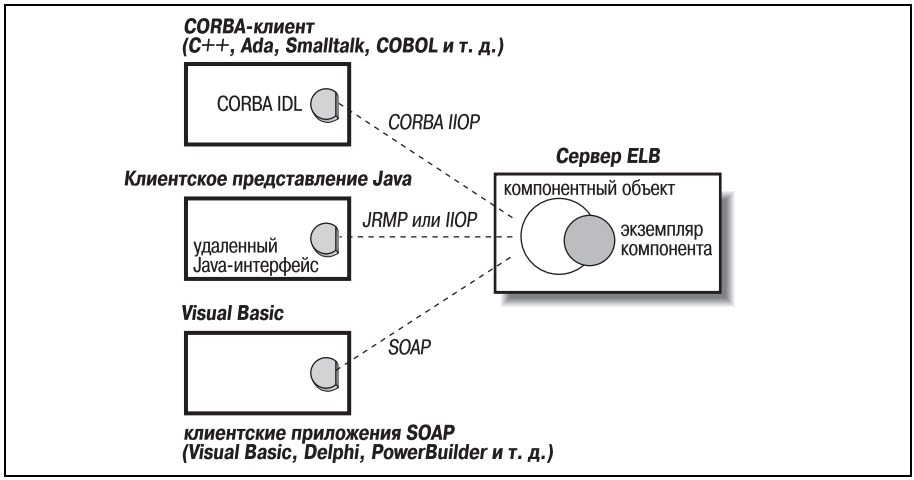

Рис. 3.13. EJB доступен для различных распределенных клиентов

Как достаточно зрелый протокол распределенных объектов, не зависящий ни от языка, ни от платформы, CORBA в настоящее время многими расценивается как наилучший из всех трех рассмотренных здесь протоколов. Однако при всех своих преимуществах CORBA имеет и некоторые ограничения. Передача по значению – особенность, легко поддерживаемая Java RMI-IIOP, только недавно была введена в спецификацию CORBA 2.3, и она еще недостаточно хорошо поддерживается производителями. Другое ограничение CORBA имеет отношение к приведению типов удаленных посредников (proxy). В Java RMI-JRMP вы можете привести или расширить удаленный интерфейс посредника к подтипу или типу базового интерфейса точно так же, как и любой другой объект. Это мощная возможность, обеспечивающая полиморфность удаленных объектов. В Java RMI-IIOP для изменения интерфейса заместителя на его подтип необходимо вызвать специальный довольно громоздкий метод сужения.[1]

Однако у JRMP есть свои собственные ограничения. Хотя этот протокол, возможно, более органичен в системах распределенных объектов типа «Java-Java», он страдает от недостатка встроенной поддержки безопасности и транзакционных сервисов, являющейся частью спецификации CORBA IIOP. Это ограничивает эффективность применения JRMP в гетерогенном окружении, где безопасность и контексты транзакций должны передаваться между системами.

EJB 2.0: Корпоративная асинхронная система передачи сообщений

До EJB 2.0 поддержка корпоративной асинхронной системы передачи сообщений, а именно службы сообщений Java, не рассматривалась в качестве базовой службы, т. к. она не являлась обязательной для законченной платформы EJB. Однако с введением в EJB 2.0 компонентов, управляемых сообщениями, корпоративная асинхронная система передачи сообщений стала настолько важной, что была повышена до базовой службы.

Поддержка этой службы достаточно сложна, но в основном она требует, чтобы контейнерная система EJB надежно переправляла сообщения от клиентов JMS компонентам, управляемым сообщениями. Это включает в себя больше чем простую семантику доставки, связанную с электронной почтой или даже JMS API. В корпоративной системе сообщения должны доставляться надежно. Это означает, что сбой при

[1] Некоторые производители в действительности поддерживают в Java-клиентах встроенное приведение типов без необходимости применения метода `RemotePortableObject.narrow()` (технически это не так сложно), но в удаленном клиентском API спецификация EJB требует использования и поддержки метода `narrow()`.

передаче сообщения может потребовать от провайдера JMS повторной попытки передачи.[1] Более того, корпоративные сообщения могут быть постоянными. Это означает, что пока они не будут корректно доставлены клиентам, для которых они предназначались, сообщения сохраняются на диске или в базе данных. Постоянные сообщения также должны быть устойчивыми к системным сбоям. Если происходит сбой сервера EJB, эти сообщения все еще будут доступны для передачи, после того как сервер возобновит работу.

Наиболее важно то, что корпоративная система передачи сообщений является транзакционной. Это означает, что если компонент, управляемый сообщениями, потерпит неудачу при обработке сообщения, то данный сбой прервет транзакцию и вынудит контейнер EJB повторно послать сообщение другому экземпляру компонента, управляемого сообщениями.

Кроме компонентов, управляемых сообщениями, JMS-сообщения могут посылать также и сеансовые компоненты без состояния, и объектные компоненты. Посылка сообщений может быть столь же важна для Enterprise JavaBeans, как и доставка сообщений к компонентам, управляемым сообщениями, – существует тенденция к взаимосвязанному развитию поддержки обоих средств.

Интересно отметить, что семантика поддержки компонентов, управляемых сообщениями, требует тесной связи между контейнерной системой EJB и маршрутизатором сообщений JMS, из-за этого многие контейнерные системы EJB поддерживают ограниченное число провайдеров JMS. Это означает, что компоненты, управляемые сообщениями, не могут получать сообщения, приходящие от произвольного провайдера JMS или продукта MOM. Сообщения компонентам, управляемым сообщениями, будут способны доставлять только провайдеры JMS, явно поддерживаемые производителем EJB.[2]

Система имен

Все службы распределенных объектов пользуются услугами системы имен некоторого вида. Java RMI-JRMP и CORBA используют свои собственные службы имен. Все службы имен независимо от того, как они реализованы, выполняют по существу одну и ту же функцию: они пре-

[1] Большинство производителей EJB устанавливают ограничение на количество попыток повторной доставки сообщений. Если передача повторяется слишком много раз, сообщение может быть помещено в хранилище «мертвых сообщений» (dead message), где оно будет доступно администратору.

[2] Эта ситуация может измениться с развитием интерфейса Java Connector в направлении более широкой поддержки асинхронных систем связи, таких как JMS, что может сделать провайдеры JMS больше чем просто службами, подключаемыми к платформе EJB.

доставляют клиентам механизм поиска распределенных объектов и ресурсов.

Для выполнения этой задачи служба имен должна предоставить две вещи: связывание объектов и поисковый API. *Связывание объектов* (*object binding*) – это механизм сопоставления распределенного объекта с именем на естественном языке или идентификатором. Объект `CabinHomeRemote`, например, мог бы быть связан с именем «CabinHomeRemote» или «комната». Связь в действительности является указателем на определенный распределенный объект, который требуется в окружении, управляющем сотнями различных распределенных объектов (или его индексом). *Поисковый API* (*lookup API*) обеспечивает клиент механизмом взаимодействия с системой имен. Короче говоря, поисковый API позволяет клиенту соединяться с распределенной службой и запрашивать удаленную ссылку на указанный объект.

Enterprise JavaBeans обязывает использовать JNDI как поисковый API на клиентах Java. JNDI поддерживает почти все виды служб каталогов и имен. Служба каталогов – это расширенная служба имен, которая организует распределенные объекты и другие ресурсы (принтеры, файлы, серверы приложений и т. д.) в иерархические структуры и предоставляет более сложные возможности управления ими. В случае применения служб каталогов клиентам также доступны метаданные, относящиеся к распределенным объектам и другим ресурсам. Метаданные предоставляют атрибуты, описывающие объект или ресурс, и могут быть использованы для их изучения. Например, можно выполнить поиск всех лазерных принтеров, находящихся в определенном здании и поддерживающих цветную печать.

Услуги каталогов также разрешают виртуальное связывание ресурсов. Это означает, что ресурсы могут находиться там, где для них будет определено место в иерархии службы каталогов. JNDI позволяет связывать разные типы служб каталогов так, чтобы клиент мог незаметно для него передвигаться по различными типам служб. Например, клиент может, следуя связи, находящейся в каталоге Novell NetWare, переместиться на сервер EJB, что позволяет серверу быть более тесно интегрированным с другими ресурсами организации, которую он обслуживает.

Клиентские приложения Java могут использовать JNDI для инициализации подключения к серверу EJB и поиска требуемого внутреннего объекта. В следующем коде показано, как может применяться JNDI API для нахождения и получения ссылки на внутренний объект `CabinHomeRemote`:

```
javax.naming.Context jndiContext = new
javax.naming.InitialContext(properties);
Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote cabinHome = (CabinHome)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

```
Cabin cabin = cabinHome.create(382, "Cabin 333",3);
cabin.setName("Cabin 444");
cabin.setDeckLevel(4);
```

Свойства, переданные в конструктор `InitialContext`, указывают JNDI API, где следует искать сервер EJB и какой провайдер (драйвер) JNDI загрузить. Метод `Context.lookup()` указывает провайдеру JNDI имя объекта, который нужно вернуть с сервера EJB. В данном случае мы ищем внутренний интерфейс компонента Cabin. После того как у нас будет внутренний интерфейс компонента Cabin, мы сможем использовать его для создания новых кают и получения доступа к существующим.

Существует множество различных видов служб имен и каталогов. Производители EJB могут выбрать тот, который лучше всего удовлетворяет их потребностям, но все платформы EJB 2.0 должны поддерживать службу имен CORBA в дополнение к любым другим службам каталогов, которые они решат поддерживать.

Для приведения типов удаленных ссылок, получаемых с помощью JNDI, к типу интерфейса `CabinHomeRemote` спецификация Enterprise JavaBeans требует применения метода `PortableRemoteObject.narrow()`. Это описано более подробно в главах 4 и 5 и не существенно для рассматриваемого здесь материала. Данное средство не требуется, когда компоненты используют локальные компонентные интерфейсы других совмещенных с ними компонентов.

Безопасность

Серверы Enterprise JavaBeans могут поддерживать целых три вида безопасности: аутентификацию (установление подлинности), управление доступом и безопасные соединения. К Enterprise JavaBeans определенно относится только управление доступом.

Аутентификация

Если говорить коротко, аутентификация проверяет подлинность пользователя. Наиболее распространенный вид установления подлинности – отображение простого экрана входа в систему (login), который требует ввода имени пользователя и пароля. Пользователи, успешно прошедшие через систему аутентификации, могут работать в системе. Аутентификация может также выполняться на основе удостоверений личности, свайп-карт (swipe cards),[1] сертификатов безопасности и других форм идентификации. Хотя аутентификация является основной гарантией против нелегального доступа к системе, одной ее не достаточно, поскольку она не наблюдает

[1] Содержимое такой магнитной карты считывается при проведении (swiping) ею над считывающим устройством. – *Примеч. ред.*

за тем, чтобы доступ к ресурсам системы получил именно авторизированный пользователь.

Управление доступом

Управление доступом (т. е. авторизация) применяет политику безопасности, регулирующую полномочия определенного пользователя внутри системы (что именно ему позволено делать, а что нет). Управление доступом гарантирует, что пользователям будут доступны только те ресурсы, на использование которых им было дано разрешение. Управление доступом может дать доступ пользователю к подсистемам, данным и прикладным объектам или контролировать более общее поведение. Некоторым пользователям, например, может быть разрешено обновлять информацию, а другим – только просматривать данные.

Безопасные соединения

Каналы связи между клиентом и сервером часто наиболее уязвимы с точки зрения безопасности. Канал связи может быть защищен с помощью физической изоляции (например, через выделенное сетевое подключение) или шифрования данных, передаваемых между клиентом и сервером. Физическая защита соединений дорого стоит, ограниченна и практически невозможна в Интернете, поэтому мы сосредоточимся на шифровании. Когда связь защищается шифрованием, передаваемые сообщения кодируются так, что они не могут быть прочитаны или изменены неуполномоченными на это лицами. Такой подход обычно связан с обменом криптографическими ключами между клиентом и сервером. Ключи позволяют получателю сообщения декодировать и прочитать его.

Большинство серверов EJB поддерживают безопасные соединения, обычно через протокол уровня безопасных сокетов (Secure Socket Layer, SSL) и некоторый механизм аутентификации, но Enterprise JavaBeans определяет для своей модели серверных компонентов только управление доступом. Аутентификация может быть определена в последующих версиях, но безопасные соединения, возможно, никогда не будут определены, т. к. являются независимыми от спецификации EJB и протокола распределенных объектов.

Хотя аутентификация не определена в EJB, она часто реализуется с помощью JNDI API. Другими словами, клиент, применяющий JNDI, может предоставить информацию об аутентификации, используя для обращения к серверу или ресурсам на сервере программный интерфейс JNDI. Эта информация часто передается, когда клиент пытается инициализировать JNDI-подключение с сервером EJB. В следующем коде показано, как пароль клиента и имя пользователя добавляются к свойствам подключения, необходимым для получения JNDI-подключения с сервером EJB:

```
properties.put(Context.SECURITY_PRINCIPAL, userName );
properties.put(Context.SECURITY_CREDENTIALS, userPassword);
```

```
javax.naming.Context jndiContext = new
javax.naming.InitialContext(properties);
Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote cabinHome = (CabinHome)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

EJB предписывает, чтобы каждое клиентское приложение, обращающееся к системе EJB, было связано с идентификатором безопасности (security identity). Идентификатор безопасности представляет клиента в качестве пользователя или роли. Пользователь может быть человеком, сертификатом безопасности, компьютером или даже смарткартой. Обычно пользователь – это человек, идентификатор которому назначается при входе в систему. Роль представляет группировку идентификаторов и может являться чем-то вроде «менеджера», представляющего собой группу идентификаторов пользователей, которые считаются менеджерами компании.

Когда удаленный клиент входит в систему EJB, на время данной сессии ему назначается идентификатор безопасности. Идентификатор находится в базе данных или каталоге, специфичном для данной платформы или сервера EJB. Эта база данных или каталог отвечают за хранение индивидуальных идентификаторов безопасности и их членства в группах.

После того как удаленному клиентскому приложению назначен идентификатор безопасности, оно может использовать компоненты, необходимые для выполнения некоторой задачи. Сервер EJB следит за каждым клиентом и его идентификатором. Когда клиент вызывает метод компонентного интерфейса, сервер EJB вместе с вызовом метода неявно передает идентификатор клиента. Когда компонентный объект или внутренний объект принимает вызов метода, он проверяет этот идентификатор, чтобы гарантировать что клиенту разрешено вызывать этот метод.

Управление доступом на основе ролей

В Enterprise JavaBeans идентификатор безопасности представлен объектом `java.security.Principal`. В качестве идентификатора безопасности в архитектуре управления доступом EJB объект `Principal` действует как представитель пользователей, групп, организаций, смарткарт и т. д. Дескрипторы развертывания включают теги, объявляющие, каким логическим ролям к каким методам компонентов разрешается обращаться во время выполнения. Роли безопасности считаются логическими ролями, поскольку в конкретном операционном окружении они непосредственно не связаны с пользователями, группами или какими-либо другими идентификаторами безопасности. Вместо этого роли безопасности сопоставляются с реальными пользователями и группами пользователей во время развертывания компонента. Это позволяет компоненту быть переносимым. Каждый раз, когда компо-

нент разворачивается в новой системе, эти роли могут быть сопоставлены с пользователями и группами, специфичными для этого операционного окружения.

Ниже приведена часть дескриптора развертывания компонента Cabin, в котором определяются две роли безопасности: `ReadOnly` и `Administrator`:

```
<security-role>
    <description>
Этой роли разрешено выполнять любой метод на компоненте, читать и изменять
любые данные компонента.
    </description>
    <role-name>
        Administrator
    </role-name>
</security-role>

<security-role>
    <description>
Этой роли разрешено определять местоположение и читать информацию о
компоненте.
        Этой роли не разрешено изменять данные компонента.
    </description>
    <role-name>
        ReadOnly
    </role-name>
</security-role>
```

Имена ролей в этом дескрипторе не являются ни зарезервированными, ни специальными именами с предопределенными значениями, это просто логические имена, выбранные сборщиком компонента. Другими словами, имена ролей могут быть какими угодно, если вы считаете их достаточно описательными.[1]

Как роли связываются с разрешенными или запрещенными действиями? После объявления тегов `<security-role>` они могут быть связаны с методами компонента с помощью тегов `<method-permission>`. Каждый тег `<method-permission>` содержит один или несколько тегов `<method>`, идентифицирующих методы компонента, связанные с одной или несколькими логическими ролями, идентифицируемыми тегами `<role-name>`. Теги `<role-name>` должны быть сопоставлены с именами, определенными тегами `<security-role>`, показанными выше:

```
<method-permission>
    <role-name>Administrator</role-name>
```

[1] Для полного понимания XML, включая определенные правила для имен тегов и данных, см. книгу «Изучаем XML» Эрика Рея, издательство «Символ-Плюс», 2001 (Eric T. Ray «Learning XML» O'Rielly & Associates, 2001).

```
        <method>
            <ejb-name>CabinEJB</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <method-permission>
        <role-name>ReadOnly</role-name>
        <method>
            <ejb-name>CabinEJB</ejb-name>
            <method-name>getName</method-name>
        </method>
        <method>
            <ejb-name>CabinEJB</ejb-name>
            <method-name>getDeckLevel</method-name>
        </method>
        <method>
            <ejb-name>CabinEJB</ejb-name>
            <method-name>findByPrimaryKey</method-name>
        </method>
    </method-permission>
```

В первом элементе `<method-permission>` роль `Administrator` связывается со всеми методами компонента Cabin, который обозначен с помощью группового символа (*) в элементе `<method-name>` тега `<method>`. Во втором `<method-permission>` роль `ReadOnly` ограничивается доступом только к трем методам: `getName()`, `getDeckLevel()` и `findByPrimaryKey()`. Все попытки роли `ReadOnly` обратиться к методу, который не включен в элемент `<method-permission>`, вызовут исключение. Этот вид управления доступом был создан для достаточно небольших систем авторизации.

Поскольку в одном XML-дескрипторе развертывания может быть описано больше одного компонента, теги, используемые для описания разрешений методов и ролей безопасности, помещаются в специальном разделе дескриптора развертывания. Это позволяет нескольким компонентам совместно использовать одни роли безопасности. Точное расположение этих тегов и их отношения с другими разделами XML-дескриптора развертывания будет рассмотрено более подробно в главе 16.

Во время развертывания компонента человек, развертывающий компонент, должен изучить информацию в `<security-role>` и сопоставить каждую логическую роль соответствующей группе пользователей в операционном окружении. Управляющему развертыванием не надо думать о том, какие роли в какие методы будут входить. Он может положиться на описания, приведенные в тегах `<security-role>`, чтобы составить пары на основе описания логической роли. Это облегчает задачу управляющего развертыванием, который может не быть разработчиком, избавляя его от необходимости понимать, как компонент работает, для того чтобы развернуть его.

На рис. 3.14 показан один и тот же компонент, развернутый в двух разных окружениях (помеченных как X и Z). В каждом окружении группы пользователей сопоставлены с их логическими эквивалентными ролями XML-дескриптора развертывания так, чтобы определенные группы пользователей имели привилегии для доступа к определенным методам определенных компонентов.

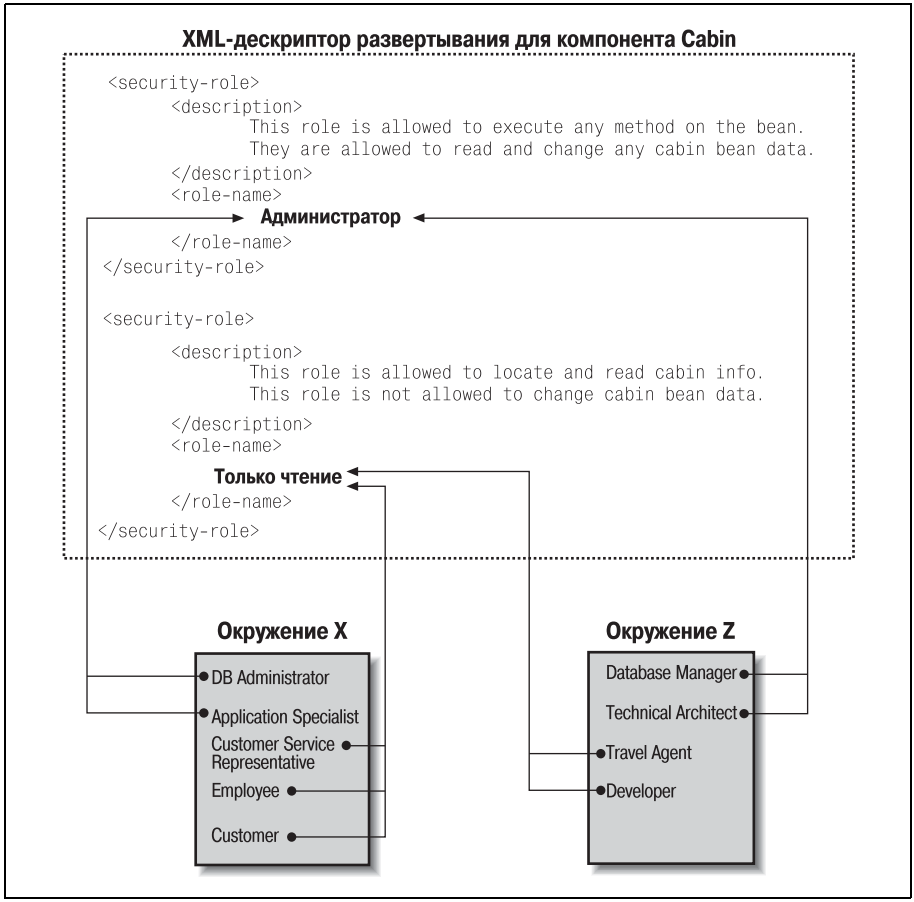

Рис. 3.14. Сопоставление ролей операционного окружения с логическими ролями дескриптора развертывания

Как видно из рисунка, роль `ReadOnly` связана с теми группами, которые следует ограничить методами доступа get и поисковым методом. Роль `Administrator` связана с теми группами пользователей, которые должны иметь привилегии для вызова любого метода компонента Cabin.

Здесь описано неявное управление доступом. После того как компонент развернут, он следит за тем, чтобы пользователи обращались

только к тем методам, на которые у них есть разрешение. Это реализуется распространением идентификатора безопасности объекта Principal на каждый вызов метода компонента со стороны клиента. Когда клиент вызывает метод компонента, принципал (объект Principal) клиента проверяется на предмет того, является ли он членом роли, связанной с этим методом. Если нет, то возбуждается исключение и клиенту отказывается в разрешении вызвать данный метод. Если клиент – член привилегированной роли, метод вызывается.

Если во время обслуживания клиента компонент пытается обратиться к любому другому компоненту, он передает идентификатор безопасности клиента для проверки контроля доступа к другим компонентам. Таким образом, клиентский Principal («принципал» клиента) распространяется от одного вызова компонента к другому, гарантируя, что его доступ контролируется, независимо от того, как он вызывает метод компонента – прямо или косвенно. В EJB 2.0 это распространение может быть отменено указанием на то, что компонент выполняется под другим идентификатором безопасности, называемым идентификатором безопасности runAs, который рассматривается ниже в этой главе.

EJB 2.0: Непроверяемые методы

В EJB 2.0 некоторые методы могут быть обозначены как *непроверяемые* (*unchecked*). Это означает, что перед вызовом такого метода разрешения безопасности не проверяются. Непроверяемый метод может быть вызван любым клиентом, независимо от того, какая роль ему сопоставлена. Для обозначения метода или методов как непроверяемых, используйте элемент <method-permission> и замените элемент <role-name> пустым элементом <unchecked>:

```
<method-permission>
    <unchecked/>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
    <method>
        <ejb-name>CustomerEJB</ejb-name>
        <method-name>findByPrimaryKey</method-name>
    </method>
</method-permission>
<method-permission>
    <role-name>administrator</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
```

Это объявление сообщает нам, что все методы компонента Cabin, а также метод findByPrimaryKey() компонента Customer, являются непроверяемыми. Хотя второй элемент <method-permission> разрешает администратору обращаться ко всем методам компонента Cabin, это объявление перекрывается разрешением непроверяемых методов. Разрешения непроверяемых методов всегда перекрывают все остальные разрешения методов.

EJB 2.0: Идентификатор безопасности runAs

Кроме определения объектов Principal (принципалов), имеющих доступ к методам компонента, управляющий развертыванием также может определить Principal идентификатора runAs для всего компонента. Идентификатор безопасности runAs первоначально был определен в EJB 1.0, но затем был отменен в EJB 1.1. Он был заново введен в EJB 2.0 и изменен так, чтобы его было проще реализовать производителям.

Тогда как элементы <method-permission> указывают, какие принципалы имеют доступ к методам компонента, элемент <security-identity> определяет, под каким принципалом метод должен выполняться. Другими словами, принципал runAs выступает в качестве идентификатора компонента, когда он пытается вызвать методы других компонентов. Этот идентификатор не обязательно является тем же идентификатором, который обращается к компоненту в настоящий момент.

Например, следующие элементы дескриптора развертывания объявляют, что к методу create() может обращаться только JimSmith, а компонент Cabin всегда выполняется под идентификатором безопасности Administrator:

```
<enterprise-beans>
...
    <entity>
        <ejb-name>EmployeeService</ejb-name>
        ...
        <security-identity>
            <run-as>
               <role-name>Administrator</role-name>
            </run-as>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
<assembly-descriptor>
<security-role>
    <role-name>Administrator</role-name>
```

```
</security-role>
<security-role>
    <role-name>JimSmith</role-name>
</security-role>
...
<method-permission>
    <role-name>JimSmith</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>create</method-name>
    </method>
</method-permission>
...
</assembly-descriptor>
```

Конфигурация такого вида полезна тогда, когда компоненты или ресурсы, доступные в теле метода, требуют принципал, отличающийся от того, который используется для получения доступа к данному методу. Например, метод `create()` мог бы вызывать метод компонента X, который требует идентификатора безопасности `Administrator`. Если необходимо использовать компонент X внутри метода create(), но мы хотим, чтобы создавать новые каюты мог только Джим Смит, нам бы следовало использовать оба элемента, `<security-identity>` и `<method-permission>`, для того чтобы дать себе такой вид гибкости: `<method-permission>` для `create()` задавал бы, что вызывать метод может только Джим Смит, а элемент `<security-identity>` определил бы, что компонент всегда выполняется под идентификатором безопасности `Administrator`. Для того чтобы задать, что компонент будет выполняться под идентификатором вызывающего объекта, роль `<security-identity>` содержит единственный пустой элемент `<use-caller-identity>`. Например, следующие объявления указывают, что компонент Cabin всегда выполняется под идентификатором вызывающего объекта, поэтому, если Джим Смит вызывает метод `create()`, компонент будет выполняться под идентификатором безопасности `JimSmith`:

```
<enterprise-beans>
...
    <entity>
        <ejb-name>CabinEJB</ejb-name>
        ...
        <security-identity>
            <use-caller-identity/>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
```

На рис. 3.15 показано, как объект `Principal` идентификатора runAs может изменяться при цепочном вызове методов. Обратите внимание, что принципал runAs – это принципал, используемый для проверки доступа при последующих вызовах методов.

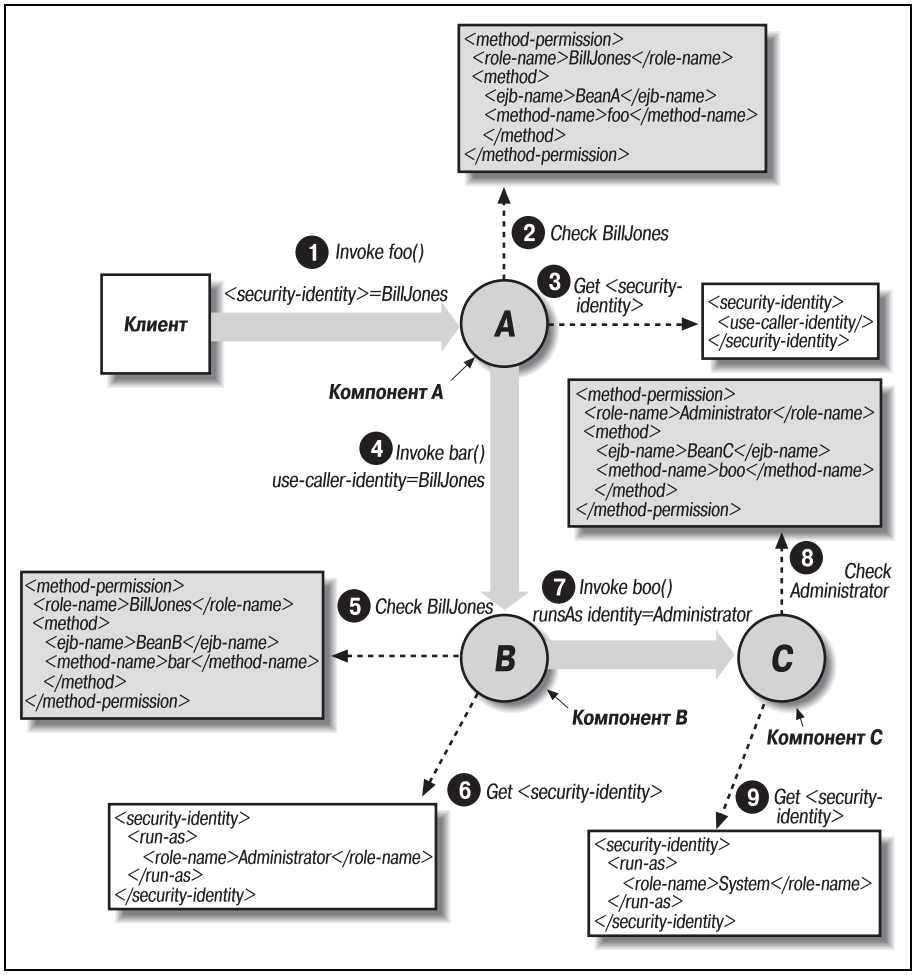

Рис. 3.15. Идентификатор runAs

Далее показано, что происходит на этом рисунке:

1. Клиент, идентифицируемый как `BillJones`, вызывает метод `foo()` компонента A.
2. Перед обслуживанием метода компонент A проверяет, включен ли `BillJones` в элементы `<method-permission>`, относящиеся к методу `foo()`. Включен.

3. <security-identity> компонента A объявлен как <use-caller-identity>, поэтому метод foo() выполняется под принципалом вызывающего объекта; в нашем случае это BillJones.
4. Во время своего выполнения метод foo() вызывает метод bar() компонента B, используя идентификатор безопасности BillJones.
5. Компонент B проверяет принципал метода foo() (BillJones) на предмет разрешенных идентификаторов для метода bar(). BillJones включен в элементы <method-permission>, поэтому методу bar() разрешается выполняться.
6. В <security-identity> для компонента B определено, что его принципал runAs должен быть установлен в Administrator.
7. Во время выполнения bar() компонент B вызывает метод boo() компонента C.
8. Компонент C проверяет, включен ли принципал runAs метода bar() (Administrator) в элементы <method-permission> для метода boo(). Включен.
9. Элемент <security-identity> для компонента C определяет принципал runAs как System, который представляет собой идентификатор, под которым выполняется метод boo().

Этот протокол применяется как к объектным компонентам, так и к сеансовым компонентам без состояния. Однако у компонентов, управляемых сообщениями, есть только идентификатор runAs. Они никогда не выполняются под идентификатором вызывающего объекта, поскольку у них нет «вызывающего объекта». Компоненты, управляемые сообщениями, обрабатывают асинхронные сообщения JMS. Эти сообщения не считаются вызовами, и клиенты JMS, посылающие их, не связаны с этими сообщениями. Без идентификатора безопасности вызывающего объекта, который нужно распространять, компоненты, управляемые сообщениями, всегда должны иметь определенный идентификатор безопасности runAs и всегда будут выполнять под этим принципалом runAs.

EJB 2.0: Базовые службы и их взаимодействие

Способность к взаимодействию является жизненно важной частью EJB 2.0. Новая спецификация включает необходимую поддержку удаленных вызовов методов для Java RMI-IIOP и обеспечивает возможность взаимодействия с транзакциями, системой имен и безопасностью.

Спецификация EJB 2.0 требует от производителей предоставления реализации Java RMI, использующей протокол IIOP CORBA 2.3.1. Цель этого требования в том, чтобы серверы J2EE были способны к взаимодействию, так, чтобы компоненты J2EE (компоненты EJB, приложе-

ния, сервлеты и JSP), находящиеся в одном сервере J2EE, могли обращаться к компонентам другого сервера J2EE. Спецификация Java RMI-IIOP стандартизирует передачу параметров, возвращаемые значения и исключения, а также отображение интерфейсов и объектов значений на CORBA IDL.

Производителям разрешается поддерживать протоколы, отличные от Java RMI-IIOP, если семантика интерфейсов RMI придерживается типов, допустимых в RMI-IIOP. Это ограничение гарантирует, что клиентское представление EJB будет согласованным независимо от протокола, используемого для удаленных вызовов.

Транзакционное взаимодействие между контейнерами для реализации двухфазной фиксации является необязательной, но важной особенностью EJB 2.0. Оно гарантирует, что транзакции, начатые вебкомпонентом J2EE, будут распространены на компоненты в других контейнерах с помощью неявного механизма распространения, описанного в службе объектных транзакций (Object Transaction Service, OTS) спецификации CORBA v1.2. Спецификация EJB 2.0 детально описывает, как обрабатываются двухфазные фиксации между несколькими контейнерами EJB, а также как транзакционные контейнеры взаимодействуют с нетранзакционными контейнерами.

В новой спецификации также затрагивается потребность во взаимодействии служб имен при поиске компонентов. В качестве взаимодействующей службы имен она указывает CORBA CosNaming, описывая, как эта служба должна реализовывать в модуле CosNaming интерфейсы IDL компонентов, и как клиенты EJB должны использовать эту службу поверх IIOP.

EJB 2.0 предоставляет безопасное взаимодействие, указывая, как контейнеры EJB устанавливают доверительные отношения и как они обмениваются сертификатами безопасности, когда компоненты J2EE обращаются к компонентам EJB из других контейнеров. От контейнеров EJB требуется поддержка протокола уровня безопасных сокетов (Secure Sockets Layer, SSL 3.0) и связанного с ним стандартного протокола безопасности транспортного уровня (Transport Layer Security, TLS 1.0) IETF для обеспечения безопасных соединений между клиентами и компонентами.

Что дальше?

Первые три главы этой книги должны были дать вам фундамент для разработки компонентов Enterprise JavaBeans и приложений. Вы должны лучше понимать CTM и компонентную модель EJB.

С главы 4 вы приступите к разработке собственных компонентов и изучите их применение в приложениях EJB.

4

Создание вашего первого компонента

Выбор и настройка сервера EJB

Одна из важнейших особенностей EJB состоит в том, что его компоненты могут выполняться в контейнерах самых разных производителей. Однако это не означает, что выбор сервера и установка на него ваших компонентов – тривиальная задача.[1]

Выбранный сервер EJB должен предоставлять средства развертывания компонентов. Не имеет значения, будут ли это средства командной строки или графические утилиты, важно только, чтобы они работали. Средства развертывания должны обеспечивать возможность работы с заранее упакованными компонентами, т. е. с компонентами, уже созданными и заархивированными в файле JAR. Наконец, сервер EJB должен поддерживать стандартные SQL-совместимые реляционные базы данных с поддержкой JDBC. Эти базы данных должны предоставлять вам привилегии, достаточные для создания и модификации нескольких простых таблиц, а также должны разрешать обычные операции чтения, записи и обновления записей. Если выбран сервер EJB, не поддерживающий реляционные базы данных SQL, то может

[1] Чтобы помочь вам в работе с продуктами от различных производителей, я написал бесплатные рабочие книги для конкретных серверов EJB. Подробную информацию об этих книгах и о том, как их достать, можно найти в предисловии.

потребоваться некоторая модификация примеров, чтобы сделать их работоспособными на выбранном продукте.

В этой книге не объясняется подробно, как устанавливать и развертывать компоненты. Эта процедура во многом зависит от сервера. Дам лишь несколько общих советов о том, как структурировать файлы JAR и создавать дескриптор развертывания, а за полным описанием процесса развертывания необходимо обратиться к документации сервера или заглянуть в рабочую книгу, относящуюся к производителю сервера.

В данной главе вам впервые представится возможность выполнить упражнение из рабочей книги. Далее в тексте вам будут встречаться *выноски* (*callouts*), указывающие на примеры рабочей книги. Выноски будут выглядеть так:

> 🕮 Рабочее упражнение 4.2. Простой сеансовый компонент

Как было сказано в предисловии, рабочие книги в формате PDF могут быть загружены с адреса *http://www.oreilly.com/catalog/entjbeans3/* или *http://www.titan-book.com*. Некоторые из этих книг могут даже быть доступны в печатном виде, и их можно будет прямо заказать по адресу *http://www.titan-books.com*.

Настройка интегрированной среды Java

Для того чтобы извлечь из этой главы как можно больше полезного, необходимо выбрать интегрированную среду разработки (IDE), включающую отладчик и позволяющую добавлять к ней файлы Java. Некоторые интегрированные среды, такие как Visual Cafe от WebGain, VisualAge от IBM, JBuilder от Borland и Forte от Sun, полностью удовлетворяют этим простым требованиям. Некоторые EJB-продукты, например WebSphere от IBM, тесно связаны с IDE, что значительно облегчает жизнь при написании, развертывании и отладке приложений.

В процессе настройки среды необходимо включить пакет `javax.ejb` с Enterprise JavaBeans. Кроме того, потребуются пакеты с JNDI, включая `javax.naming`, `javax.naming.directory` и `javax.naming.spi`, а также пакеты `javax.rmi` и `javax.jms`. Все эти пакеты можно загрузить с сайта Sun Java (*http://www.javasoft.com*) в виде файлов ZIP или JAR. Их можно найти и в подкаталогах вашего EJB-сервера, обычно в каталоге *lib*.

Создание объектного компонента

Мне кажется, что лучше всего было бы начать с компонента Cabin, который рассматривался на протяжении всей предыдущей главы. Компонент Cabin – это объектный компонент, включающий в себя данные и поведение связанной с ним корабельной каюты в предприятии «Титан».

Cabin: Удаленный интерфейс

При создании объектного компонента первым делом необходимо определить удаленный интерфейс компонента. Удаленный интерфейс определяет прикладное назначение компонента. Методы этого интерфейса должны охватывать понятие, связанное с компонентом. Мы определили удаленный интерфейс компонента Cabin в главе 2. Здесь же добавим два новых метода для установки и получения идентификатора (ID) корабля и количества коек. Идентификатор корабля определяет корабль, которому принадлежит данная каюта, а количество коек – число пассажиров, в каюте размещающихся.

```
package com.titan.cabin;

import java.rmi.RemoteException;

public interface CabinRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String str) throws RemoteException;
    public int getDeckLevel() throws RemoteException;
    public void setDeckLevel(int level) throws RemoteException;
    public int getShipId() throws RemoteException;
    public void setShipId(int sp) throws RemoteException;
    public int getBedCount() throws RemoteException;
    public void setBedCount(int bc) throws RemoteException;
}
```

Интерфейс `CabinRemote` определяет четыре свойства: `name`, `deckLevel`, `shipId` и `bedCount`. *Свойства* (*properties*) – это атрибуты компонента, доступ к которым обеспечивают открытые методы set и get. Методы, обращающиеся к свойствам, не определены детально в интерфейсе `CabinRemote`, но интерфейс однозначно указывает, что эти атрибуты разрешены для чтения и записи клиентом.

Заметьте, что мы сделали интерфейс `CabinRemote` частью нового пакета, названного `com.titan.cabin`. Размещайте все классы и интерфейсы, относящиеся к какому-либо типу компонента, в отдельные пакеты, отведенные для этого компонента.[1] По той причине, что все наши компоненты предназначены для использования в круизах «Титан», мы поместили эти пакеты в структуру пакета `com.titan`. Мы также создали структуру каталогов, соответствующую структуре пакетов. Если вы имеете дело со средой разработки, напрямую обращающейся к файлам Java, создайте где-нибудь новый каталог с названием *dev* (for develop-

[1] Примеры, которые могут быть загружены с *http://www.orielly.com/catalog/entjbeans/*, представляют хорошее руководство по организации кода. Код имеет структуру каталогов, являющуюся типовой для большинства продуктов. Эти упражнения предоставят дополнительную помощь по организации рабочих проектов и укажут на все требования, предъявляемые конкретным производителем.

ment – для разработки) и создайте в нем структуру каталогов, изображенную на рис. 4.1. Скопируйте интерфейс `CabinRemote` в свою среду разработки и сохраните его описание в каталоге *cabin*. Откомпилируйте интерфейс `CabinRemote` для того, чтобы убедиться, что его определение не содержит ошибок. Сгенерированный компилятором файл *CabinRemote.class* должен быть записан в каталог *cabin* – тот же каталог, где находится и файл *CabinRemote.java*. Все остальные классы компонента Cabin тоже будут помещены в этот же каталог.

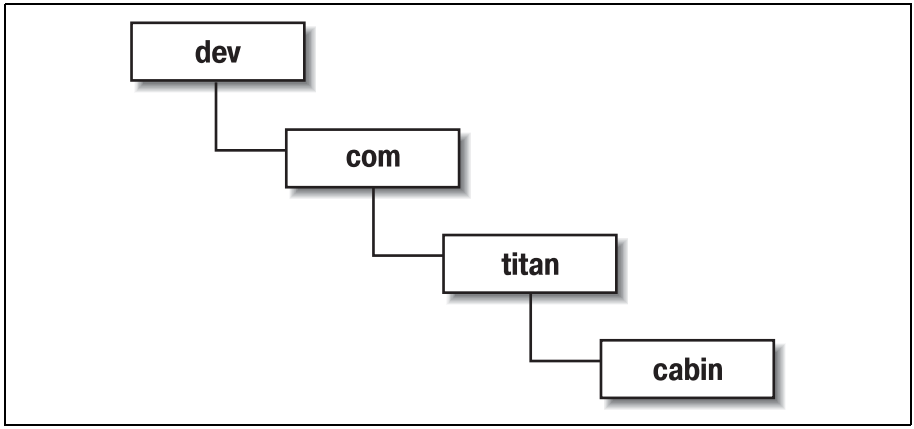

Рис. 4.1. Структура каталогов для компонента Cabin

CabinHome: Удаленный внутренний интерфейс

Определив удаленный интерфейс, мы задали удаленное представление нашего объектного компонента. Затем нам надо определить для компонента Cabin удаленный внутренний интерфейс, определяющий то, как компонент будет создаваться, находиться и уничтожаться удаленными клиентами, другими словами – характер жизненного цикла. Далее приведено полное описание внутреннего интерфейса `CabinHomeRemote`:

```
package com.titan.cabin;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHomeRemote extends javax.ejb.EJBHome {

    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;

    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

Интерфейс CabinHomeRemote расширяет javax.ejb.EJBHome и определяет два метода жизненного цикла: create() и findByPrimaryKey(). Эти методы отвечают за создание и поиск ссылок на компонент Cabin. Удаляющие (remove) методы (для уничтожения компонентов) определены в интерфейсе javax.ejb.EJBHome, поэтому интерфейс CabinHomeRemote просто наследует их.

CabinBean: Класс компонента

К настоящему моменту мы полностью определили клиентский интерфейс для создания, поиска, удаления и применения компонента Cabin. Сейчас нам необходимо определить CabinBean – класс, предоставляющий серверную реализацию компонента Cabin. Класс CabinBean – это объектный компонент, использующий постоянство, управляемое контейнером, поэтому его определение будет достаточно простым.

В дополнение к методам обратного вызова, рассмотренным в главах 2 и 3, мы также должны определить для интерфейса CabinRemote методы доступа и реализовать метод create(), определенный в интерфейсе CabinHomeRemote.

На протяжении всей книги мы будем рассматривать оба варианта кода и для EJB 2.0, и для EJB 1.1 в тех местах, где они отличаются. Во многих случаях компонентные интерфейсы идентичны, но класс компонента и дескриптор развертывания будут отличаться. Это, например, происходит в случае с компонентом Cabin, поэтому для него приводятся два листинга: первый – для EJB 2.0, а второй – для EJB 1.1. Те, кто работает с EJB 2.0, могут пропустить листинг для EJB 1.1 и наоборот.

EJB 2.0: Класс CabinBean

Здесь приведено полное определение класса CabinBean для EJB 2.0:

```
package com.titan.cabin;

import javax.ejb.EntityContext;

public abstract class CabinBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        this.setId(id);
        returns null;
    }
    public void ejbPostCreate(Integer id){

    }
    public abstract void setId(Integer id);
    public abstract Integer getId();

    public abstract void setShipId(int ship);
    public abstract int getShipId();
```

```
    public abstract void setName(String name);
    public abstract String getName();

    public abstract void setBedCount(int count);
    public abstract int getBedCount();

    public abstract void setDeckLevel(int level);
    public abstract int getDeckLevel();

    public void setEntityContext(EntityContext ctx) {
        // Не реализован
    }
    public void unsetEntityContext() {
        // Не реализован
    }
    public void ejbActivate() {
        // Не реализован
    }
    public void ejbPassivate() {
        // Не реализован
    }
    public void ejbLoad() {
        // Не реализован
    }
    public void ejbStore() {
        // Не реализован
    }
    public void ejbRemove() {
        // Не реализован
    }
}
```

Для удобства обсуждения класс `CabinBean` можно разбить на четыре части: объявления полей, управляемых контейнером, методы `ejbCreate()` и `ejbPostCreate()`, методы обратного вызова и реализацию удаленного интерфейса.

В CabinBean определены несколько абстрактных методов доступа, существующих парами. Например, методы `setName()` и `getName()` образуют одну пару абстрактных методов. Они отвечают за установку и получение поля компонента `name`. Во время развертывания этого компонента контейнер EJB автоматически реализует все абстрактные методы доступа таким образом, чтобы состояние компонента было синхронизировано с базой данных. Эти реализации связывают абстрактные методы доступа с полями базы данных. Хотя у всех абстрактных методов доступа есть соответствующие им методы в удаленном интерфейсе `CabinRemote`, это не является обязательным. Некоторые методы доступа предназначены только для внутреннего использования в компоненте и никогда не выставляются клиенту через удаленные или локальные интерфейсы.

В EJB 2.0 принято рассматривать абстрактные методы в качестве средства доступа к виртуальным полям и ссылаться на эти поля по именам их методов без префиксов get и set. Например, абстрактные методы доступа `getName()` и `setName()` определяют виртуальное поле *постоянства, управляемого контейнером* (*Container-Managed Persistence, CMP*), с именем `name` (первая буква всегда преобразуется к прописной). Методы доступа `getDeckLevel()` и `setDeckLevel()` определяют виртуальное поле CMP с именем `deckLevel` и т. д.

Виртуальные поля CMP `name`, `deckLevel`, `shipId` и `bedCount` представляют постоянное состояние компонента Cabin. Во время развертывания они будут связаны с базой данных. Эти поля являются общедоступными через удаленный интерфейс компонента. Вызов метода `getBedCount()` объекта `CabinRemote` приводит к перенаправлению контейнером этого вызова соответствующему методу экземпляра `CabinBean`. В отличие от связывания методов с удаленным интерфейсом, абстрактные методы доступа не генерируют исключений `RemoteException`.

Поля CMP не обязаны быть общедоступными. Поле `id` – еще одно поле, управляемое контейнером, но его абстрактные методы доступа не выставляются клиентам через интерфейс `CabinRemote`. Оно представляет собою первичный ключ компонента Cabin, это указатель на данные компонента в базе данных. Выставление первичного ключа компонента – плохая привычка, т. к. нельзя давать клиентским приложениям возможность изменять этот ключ.

EJB 1.1: Класс CabinBean

Здесь приведено полное определение класса `CabinBean` для EJB 1.1:

```
package com.titan.cabin;

import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public Integer id;
    public String name;
    public int deckLevel;
    public int shipId;
    public int bedCount;

    public Integer ejbCreate(Integer id) {
        this.id = id;
        return null;
    }
    public void ejbPostCreate(Integer id) {
        // Ничего не делает, но необходим
    }
    public String getName() {
        return name;
    }
```

```
    public void setName(String str) {
        name = str;
    }
    public int getShipId() {
        return shipId;
    }
    public void setShipId(int sp) {
        shipId = sp;
    }
    public int getBedCount() {
        return bedCount;
    }
    public void setBedCount(int bc) {
        bedCount = bc;
    }
    public int getDeckLevel() {
        return deckLevel;
    }
    public void setDeckLevel(int level ) {
        deckLevel = level;
    }

    public void setEntityContext(EntityContext ctx) {
         // Не реализован
    }
    public void unsetEntityContext() {
         // Не реализован
    }
    public void ejbActivate() {
        // Не реализован
    }
    public void ejbPassivate() {
        // Не реализован
    }
    public void ejbLoad() {
        // Не реализован
    }
    public void ejbStore() {
        // Не реализован
    }
    public void ejbRemove() {
        // Не реализован
    }
}
```

Поля, объявленные в классе компонента, могут быть *постоянными* (*persistent*) полями или полями *свойств* (*properties*). Эти категории не являются взаимоисключающими. Объявления постоянных полей описывают поля, которые будут связаны с базой данных. Постоянное поле часто является свойством (с точки зрения JavaBeans): любой атрибут, доступный через открытые методы set и get. Конечно, компонент мо-

жет иметь любые поля, которые ему требуются, и они не обязательно должны быть постоянными или полями-свойствами. Поля, не являющиеся постоянными, не будут сохранены в базе данных. В классе CabinBean все поля постоянные.

Поле id постоянное, но это не свойство. Другими словами, id связано с базой данных, но не может быть доступно через удаленный интерфейс.

Поля name, deckLevel, shipId и bedCount также являются постоянными. Во время развертывания они будут связаны с базой данных. Эти поля являются также и свойствами, т. к. они открыто доступны через удаленный интерфейс.

EJB 2.0 и 1.1: Методы обратного вызова

В случае с компонентом Cabin имелся только один метод create(), поэтому существует только один соответствующий ему метод ejbCreate() и один метод ejbPostCreate(). Метод create() удаленного внутреннего интерфейса, вызванный клиентом, перенаправляется соответствующему методу ejbCreate() экземпляра объектного компонента. Метод ejbCreate() инициализирует поля. В случае с CabinBean он устанавливает поле name.

Метод ejbCreate() всегда возвращает значение с типом первичного ключа. При использовании постоянства, управляемого контейнером, этот метод возвращает значение null. Ответственность за создание первичного ключа лежит на контейнере. Почему он возвращает null? Если говорить коротко, то это соглашение упрощает компоненту, управляющему собой самостоятельно (т. е. компоненту, который явно управляет своим собственным постоянством), задачу расширения компонента, управляемого контейнером. Эти функциональные возможности полезны для тех производителей EJB, которые поддерживают компоненты с постоянством, управляемым контейнером, путем расширения их с помощью реализаций компонентов с постоянством, реализуемым компонентом. Эта техника больше распространена в EJB 1.1. Компоненты с постоянством, реализуемым компонентом, рассматриваемые в главе 10, всегда возвращают значение с типом первичного ключа.

После завершения метода ejbCreate() для выполнения всех последующих действий вызывается метод ejbPostCreate(). Методы ejbCreate() и ejbPostCreate() должны иметь сигнатуры, соответствующие параметрам и (дополнительно) исключениям метода create() внутреннего интерфейса. Метод ejbPostCreate() применяется для выполнения любой постобработки компонента после того, как он создан, но перед тем как он может быть использован клиентом. Оба метода будут выполняться один за другим, когда клиент вызовет метод create() удаленного внутреннего интерфейса.

Метод findByPrimaryKey() не определяется в классах компонента, управляемого контейнером. Вместо этого поисковые методы генериру-

ются во время развертывания и реализуются контейнером. В объектных компонентах, управляемых компонентом, поисковые методы должны быть определены в классе компонента. В главе 10, где рассматривается разработка объектных компонентов, управляемых компонентом, мы будем определять поисковые методы внутри создаваемых классов компонентов.

Класс CabinBean реализует `javax.ejb.EntityBean`, в котором определены семь методов обратного вызова: `setEntityContext()`, `unsetEntityContext()`, `ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, `ejbStore()` и `ejbRemove()`. Контейнер использует эти методы обратного вызова для уведомления CabinBean о возникновении определенных событий в его жизненном цикле. Хотя методы обратного вызова реализованы, их реализации – пустые. CabinBean довольно прост и не должен выполнять какую-либо специальную обработку во время своего жизненного цикла. Когда мы будем более подробно изучать объектные компоненты в главах 6–11, то воспользуемся этими методами обратного вызова.

Дескриптор развертывания

Итак, мы готовы создать дескриптор развертывания для компонента Cabin. Дескриптор развертывания служит тем же целям, что и файл свойств. Он описывает, какие классы составляют компонент и как необходимо управлять компонентом во время выполнения. Во время процесса развертывания дескриптор развертывания считывается, а его свойства отображаются для редактирования. Затем управляющий развертыванием может изменить и добавить параметры настройки, требуемые оперативному окружению приложения. Если управляющий развертыванием удовлетворен информацией о развертывании, то использует ее для генерирования всей вспомогательной инфраструктуры, необходимой для развертывания компонента в сервере EJB. Это может быть связано с разрешением компонентных ссылок, добавлением компонента к системе имен и генерированием компонентного и внутреннего объектов, инфраструктуры постоянства, поддержки транзакций и т. д.

Хотя большинство серверов EJB предоставляют мастер для создания и редактирования дескрипторов развертывания, наш мы будем создавать непосредственно – так, чтобы компонент был определен способом, независимым от конкретного производителя.[1] Это требует некоторой ручной работы, зато вы лучше поймете, как создаются дескрипторы развертывания. Закончив работу над дескриптором развертывания, компонент можно поместить в файл JAR и развернуть на любом EJB-совместимом сервере подходящей версии.

[1] В рабочих книгах показано, как следует использовать инструменты, предоставленные производителем, для создания дескрипторов развертывания.

XML-дескрипторы развертывания для всех примеров в этой книге уже написаны и доступны для загрузки с сайта.

В следующих разделах дается краткий обзор дескрипторов развертывания в EJB 2.0 и 1.1 для компонента Cabin. Это поможет вам понять, как структурирован XML-дескриптор развертывания и какого типа информация в нем содержится.

EJB 2.0: Дескриптор развертывания компонента Cabin

В EJB 2.0 дескриптор развертывания выглядит так:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.cabin.CabinHomeRemote</home>
            <remote>com.titan.cabin.CabinRemote</remote>
            <ejb-class>com.titan.cabin.CabinBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <abstract-schema-name>Cabin</abstract-schema-name>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>deckLevel</field-name></cmp-field>
            <cmp-field><field-name>shipId</field-name></cmp-field>
            <cmp-field><field-name>bedCount</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: Дескриптор развертывания компонента Cabin

А так дескриптор развертывания выглядит в EJB 1.1:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.cabin.CabinHomeRemote</home>
            <remote>com.titan.cabin.CabinRemote</remote>
```

```
            <ejb-class>com.titan.cabin.CabinBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>deckLevel</field-name></cmp-field>
            <cmp-field><field-name>shipId</field-name></cmp-field>
            <cmp-field><field-name>bedCount</field-name></cmp-field>
            <primkey-field>id</primkey-field>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

EJB 2.0 и 1.1: Определение элементов XML

Элемент `<!DOCTYPE>` описывает назначение файла XML, его корневой элемент и местоположение его DTD. DTD предназначено для проверки правильности структуры документа. Этот элемент рассматривается подробно в главе 16. Одно важное различие между EJB 2.0 и EJB 1.1 состоит в том, что они используют для дескрипторов развертывания разные DTD. EJB 2.0 определяет `ejb-jar_2_0.dtd`, а EJB 1.1 определяет `ejb-jar_1_1.dtd`.

Остальные элементы XML вкладываются один в другой и разделяются начальными и конечными тегами. Структура не сложная. Те, кто когда-нибудь кодировал HTML, уже должны понимать этот формат. Элемент всегда начинается с тега *<имя_тега>*, а заканчивается тегом *</имя_тега>*. Все, что находится между ними (даже другие элементы), является частью заключающего элемента.

Первый главный элемент – это элемент `<ejb-jar>`, являющийся корнем документа. Все остальные элементы должны находиться внутри этого элемента. Затем идет элемент `<enterprise-beans>`. Каждый компонент, описываемый в файле XML, должен быть помещен в этот раздел. Данный файл описывает только компонент Cabin, но в одном дескрипторе развертывания мы можем определить несколько компонентов.

Элемент `<entity>` показывает, что компоненты, определенные внутри этого тега, представляют собой объектные компоненты. Точно так же элемент `<session>` описывает сеансовые компоненты. Поскольку компонент Cabin является объектным компонентом, элемент `<session>` нам не нужен. Кроме описания элемент `<entity>` предоставляет полные имена классов удаленного интерфейса, внутреннего интерфейса, класса компонента и первичного ключа. В элементах `<cmp-field>` перечисляются все управляемые контейнером поля класса объектного компонента. Это те поля, которые будут сохраняться в базе данных и управ-

ляться контейнером во время выполнения. Элемент <entity> также включает элемент <reentrant>, который может быть установлен в True или False в зависимости от того, допускает ли компонент повторные вхождения.

EJB 2.0 определяет имя name, используемое в языке запросов EJB (Query Language, QL) для идентификации объектного компонента. Сейчас это пока не важно. Дескриптор развертывания 2.0 также определяет <security-identity> как <use-caller-identity/>, который просто означает, что компонент при обращении к ресурсам или другим компонентам будет распространять идентификатор безопасности вызывающего клиента. Это было подробно рассмотрено в главе 3.

Раздел файла XML после элемента <enterprise-beans> заключен в элемент <assembly-descriptor>, который описывает роли безопасности и атрибуты транзакции компонента. Содержание этого раздела файла XML в данном примере совпадает и для EJB 2.0, и для EJB 1.1:

```
<ejb-jar>
    <enterprise-beans>
    ...
    <enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <description>
Эта роль представляет всех, кто может получить полный доступ к Cabin EJB.
            </description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Может показаться странным отделение информации в <assembly-descriptor> от информации в <enterprise-beans>, поскольку ясно, что она применяется к компоненту Cabin, но по существу это совершенно естественно. В одном XML-дескрипторе развертывания может быть опи-

сано несколько компонентов, каждый из них может использовать одни и те же роли безопасности и атрибуты транзакции. Для облегчения развертывания сразу нескольких компонентов вся эта общая информация сгруппирована в элементе <assembly-descriptor>.

Существует другая (возможно, более важная) причина для отделения информации, относящейся непосредственно к компоненту, от ролей безопасности и атрибутов транзакции. Enterprise JavaBeans определяет обязанности разных участников процесса создания и развертывания компонентов. Мы не описываем в данной книге эти роли процесса разработки, поскольку они не жизненно важны при изучении основных принципов EJB. Пока достаточно знать, что человек, который разрабатывает компоненты, и человек, который собирает компоненты в приложение, имеют разные обязанности и поэтому работают с разными частями XML-дескриптора развертывания. Разработчик компонентов отвечает за все, что находится внутри элемента <enterprise-beans>, а сборщик компонентов отвечает за все, что находится внутри <assembly-descriptor>. Читая данную книгу, вы будете выступать в обеих этих ролях, разрабатывая компоненты и собирая их. А вот другие роли, которые вам придется «играть»: управляющий развертыванием – человек, который реально загружает компоненты в контейнер EJB, и администратор, который отвечает за настройку сервера EJB и управление им во время выполнения. В реальных проектах все эти роли могут выполняться одним или двумя людьми или несколькими разными лицами и даже группами.

<assembly-descriptor> содержит элементы <security-role> и соответствующие им элементы <method-permission>, которые были описаны в разделе «Безопасность» главы 3. В этом примере имеется одна роль безопасности, everyone, которая сопоставлена со всеми методами компонента Cabin с помощью элемента <method-permission> (звездочка в элементе <method-name> означает «все методы»). Как уже отмечалось, в EJB 2.0 необходимо указать идентификатор безопасности. В данном случае это идентификатор вызывающего объекта.

Элемент <container-transaction> объявляет, что все методы компонента Cabin имеют атрибут транзакции Required. Это означает, что все методы должны выполняться в пределах транзакции. Атрибуты транзакции более подробно рассмотрены в главе 14. Дескриптор развертывания завершается закрывающим тегом элемента <ejb-jar>.

Скопируйте дескриптор развертывания для компонента Cabin в тот же каталог, где находятся файлы классов компонента Cabin (*Cabin.class*, *CabinHome.class*, *CabinBean.class* и *CabinPK.class*), и сохраните его как *ejb-jar.xml*. Итак, созданы все файлы, необходимые для того, чтобы упаковать наш компонент Cabin. На рис. 4.2 показаны все файлы, которые должны присутствовать в каталоге *cabin*.

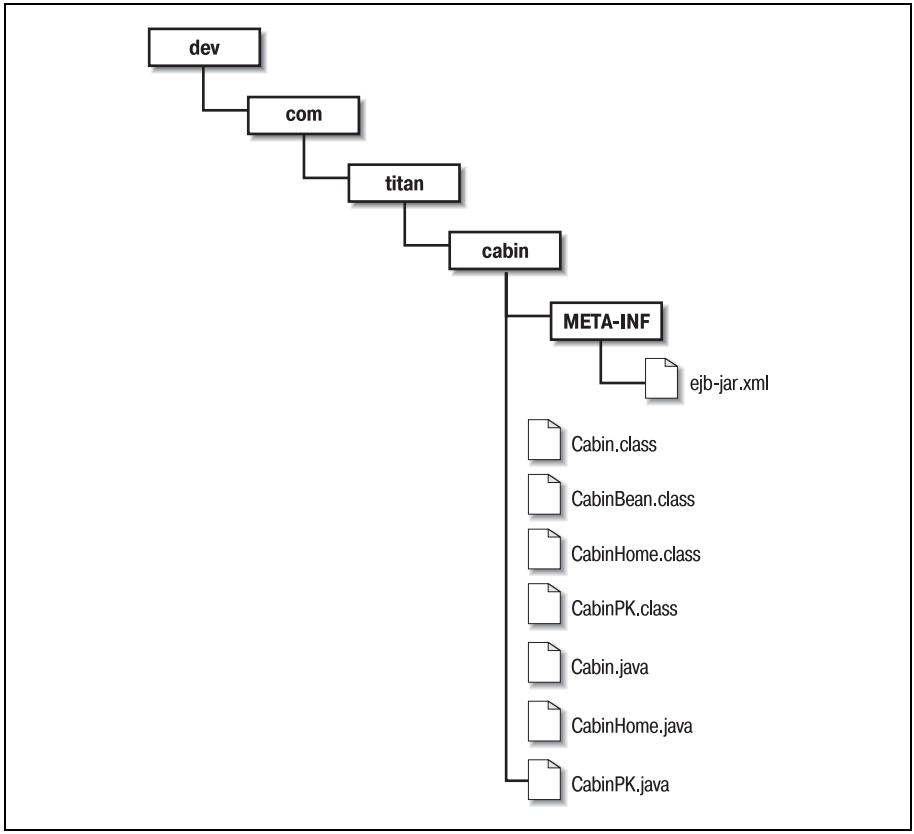

Рис. 4.2. Файлы компонента Cabin

cabin.jar: Файл JAR

Файл JAR – это независимый от платформы формат файла для сжатия, упаковки и распространения нескольких файлов совместно. Основанные на формате файла ZIP и стандартах сжатия ZLIB, средства и пакеты JAR (Java archive – архив Java) были изначально разработаны для обеспечения более эффективной загрузки апплетов Java. Однако формат файла JAR оказался очень удобным способом совместной упаковки компонентов и другого программного обеспечения для распространения третьим лицам. Оригинальная компонентная архитектура JavaBeans при упаковке использует файлы JAR, то же самое справедливо и для Enterprise JavaBeans. Цель применения в EJB формата файлов JAR состоит в том, чтобы упаковать в один файл все классы и интерфейсы, связанные с данным компонентом, включая дескриптор развертывания.

Создание файла JAR для развертывания достаточно просто. Перейдите в каталог *dev*, расположенный выше дерева каталогов *com/titan/cabin*, и выполните следующую команду:

```
\dev % jar cf cabin.jar com/titan/cabin/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\cabin\*.class META-INF\ejb-jar.xml
```

Возможно, сначала придется создать каталог *META-INF* и скопировать в него *ejb-jar.xml*. Ключ *c* указывает утилите *jar* на необходимость создать новый файл JAR, который будет содержать файлы, указанные в последующих параметрах. Он также сообщает утилите *jar*, что итоговый файл JAR должен быть направлен на стандартный вывод. Если задан ключ *f*, то *jar* перенаправит стандартный вывод в новый файл, заданный вторым параметром (*cabin.jar*). Необходимо точно воспроизвести порядок указания ключей и параметров командной строки. Об утилите *jar* и пакете `java.util.zip` можно подробнее узнать из книг издательства O'Reilly «Java in a Nutshell» (Java. Справочник) Дэвида Флэнагана (David Flanagan) и «Learning Java» (Изучаем Java) Пата Нимайера и Джонатана Нудсена (Pat Nimaier & Jonathan Knudsen).

Утилита *jar* создала в каталоге *dev* файл *cabin.jar*. Те, кто заинтересуется содержанием файла JAR, могут воспользоваться любым стандартным приложением ZIP (WinZip, PKZIP и т. д.) или выполнить команду *jar tvf cabin.jar*.

Создание таблицы CABIN в базе данных

Одной из основных задач средства развертывания является отображение объектных компонентов на базы данных. В случае с компонентом Cabin нам необходимо сопоставить его управляемые контейнером поля `id`, `name`, `deckLevel`, `shipId` и `bedCount` с некоторым источником данных. Прежде чем продолжить развертывание, необходимо установить базу данных и создать таблицу `CABIN`, совместимую с примерами этой главы. Последнее может быть сделано посредством следующего стандартного оператора SQL:

```
create table CABIN
(
    ID int primary key NOT NULL,
    SHIP_ID int,
    BED_COUNT int,
    NAME char(30),
    DECK_LEVEL int
)
```

Этот оператор создает таблицу `CABIN`, содержащую пять столбцов, соответствующих полям, управляемым контейнером, класса `CabinBean`. Создав таблицу и подтвердив связь с базой данных, можно продолжить процесс развертывания.

Развертывание компонента Cabin

Развертывание – это процесс считывания файла JAR компонента, изменения или добавления свойств к дескриптору развертывания, связывания компонента с базой данных, задания управления доступом в области безопасности и генерирования всех специфичных для данного производителя классов, необходимых для поддержки компонента в окружении EJB. Каждый сервер EJB имеет свои собственные инструментальные средства развертывания, которые могут предоставлять графический интерфейс пользователя, набор программ командной строки или и то и другое. Проще всего работать с графическими «мастерами» развертывания.

Средство развертывания читает архив JAR и ищет в нем файл *ejb-jar.xml*. В графическом мастере развертывания элементы дескриптора развертывания представляются в виде набора окон свойств, похожих на те, которые используются для настройки визуальных компонентов в средах Visual Basic, PowerBuilder, JBuilder и Symantec Cafe. На рис. 4.3 изображен мастер развертывания, применяемый в сервере J2EE 1.3 SDK (эталонная реализация).

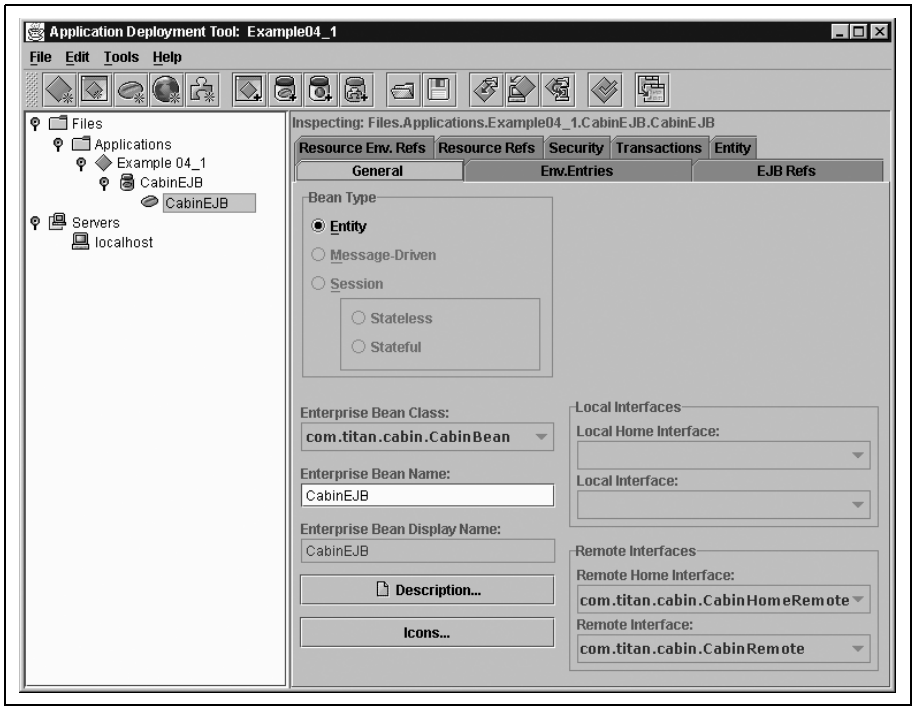

Рис. 4.3. Мастер развертывания эталонной реализации J2EE 1.3 SDK

Мастер развертывания эталонной реализации J2EE содержит поля и панели, соответствующие XML-дескриптору развертывания. Можно

связать роли безопасности с группами пользователей, установить имена поиска JNDI, сопоставить управляемые контейнером поля с базой данных и т. д.

Разные инструментальные средства развертывания EJB предоставляют различную степень поддержки для отображения управляемых контейнером полей на источник данных. Некоторые из них предоставляют очень мощные и сложные графические интерфейсы, тогда как другие – более простые и менее гибкие. К счастью, отображение управляемых контейнером полей класса `CabinBean` на таблицу `CABIN` – достаточно простая процедура. Как выполнить это сопоставление, можно узнать в документации по тому или иному средству развертывания. Закончив отображение, можно завершать развертывание компонента Cabin и готовиться к обращению к нему со стороны сервера EJB.

Создание клиентского приложения

Итак, компонент Cabin развернут на сервере EJB, и нам необходимо обратиться к нему со стороны удаленного клиента. Говоря «удаленный», мы обычно имеем в виду клиентское приложение, расположенное или на другом компьютере, или в другом процессе на том же компьютере. В этом разделе мы создадим удаленный клиент, который соединится с сервером EJB, найдет удаленный внутренний объект, относящийся к компоненту Cabin, и создаст несколько компонентов Cabin и поработает с ними. В следующем фрагменте кода показано приложение Java, предназначенное для создания нового компонента Cabin, установления его свойств `name`, `deckLevel`, `shipId` и `bedCount` и последующего повторного его поиска по первичному ключу:

```
package com.titan.cabin;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class Client_1 {
    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup("CabinHomeRemote");
            CabinHomeRemote home = (CabinHomeRemote)
                PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
            CabinRemote cabin_1 = home.create(new Integer(1));
            cabin_1.setName("Master Suite");
            cabin_1.setDeckLevel(1);
```

```
            cabin_1.setShipId(1);
            cabin_1.setBedCount(3);

            Integer pk = new Integer(1);

            CabinRemote cabin_2 = home.findByPrimaryKey(pk);
            System.out.println(cabin_2.getName());
            System.out.println(cabin_2.getDeckLevel());
            System.out.println(cabin_2.getShipId());
            System.out.println(cabin_2.getBedCount());

        } catch (java.rmi.RemoteException re){re.printStackTrace();}
          catch (javax.naming.NamingException ne){ne.printStackTrace();}
          catch (javax.ejb.CreateException ce){ce.printStackTrace();}
          catch (javax.ejb.FinderException fe){fe.printStackTrace();}
    }

    public static Context getInitialContext()
        throws javax.naming.NamingException {

        Properties p = new Properties();
        // ... задаем свойства JNDI, специфичные для производителя
        return new javax.naming.InitialContext(p);
    }
}
```

Клиент, собирающийся обратиться к компоненту, начинает с использования пакета JNDI для получения соединения с контейнером компонента. JNDI – это независимый от реализации API для систем имен и каталогов. Каждый производитель EJB должен предоставить JNDI-совместимую службу каталогов. А именно, провайдер услуг JNDI, являющийся частью программного обеспечения, аналогично драйверу в JDBC. Разные провайдеры услуг соединяются с разными службами каталогов. Это мало чем отличается от JDBC, где разные драйверы соединяются с разными реляционными базами данных. Метод `getInitialContext()` содержит логику, основывающуюся на JNDI для получения сетевого соединения с сервером EJB.

Код, применяемый для получения JNDI `Context`, будет зависеть от того, с сервером какого производителя EJB вы работаете. Ознакомьтесь с документацией на сервер, чтобы выяснить, как получить JNDI `Context` в вашем продукте. Например, код, используемый для получения JNDI `Context` в WebSphere, мог бы выглядеть так:

```
public static Context getInitialContext()
    throws javax.naming.NamingException {

    java.util.Properties properties = new java.util.Properties();
    properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
    return new InitialContext(properties);
}
```

Тот же самый метод, разработанный для сервера WebLogic от BEA, будет другим:

```
public static Context getInitialContext()
    throws javax.naming.NamingException {

    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
     "weblogic.jndi.WLInitialContextFactory");
    p.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new javax.naming.InitialContext(p);
}
```

После установления JNDI-соединения и получения контекста из метода `getInitialContext()` этот контекст может использоваться для поиска внутреннего объекта компонента Cabin.

В приложении `Client_1` применяется метод `PortableRemoteObject.narrow()`:

```
Object ref = jndiContext.lookup("CabinHome");
CabinHome home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
```

Метод `PortableRemoteObject.narrow()` первоначально был введен в EJB 1.1 и продолжает применяться на удаленных клиентах в EJB 2.0. Он нужен для поддержки требований RMI поверх IIOP. Из-за того что CORBA поддерживает множество различных языков, приведение не присуще CORBA (в некоторых языках нет приведения). Поэтому для получения удаленной ссылки на `CabinHomeRemote` мы должны явно сужать объект, возвращаемый методом `lookup()`. Результат тот же, что и для обычного приведения; более подробно это объясняется в главе 5.

Имя, по которому выполняется поиск внутреннего объекта компонента Cabin, задается во время развертывания с помощью мастера развертывания, о чем говорилось выше. JNDI-имя выбирается на усмотрение лица, развертывающего компонент. Оно может совпадать с именем компонента, установленным в XML-дескрипторе развертывания, или быть чем-то совершенно другим.

Создание нового компонента Cabin

Получив удаленную ссылку на внутренний объект, мы можем использовать ее для создания нового объекта Cabin:

```
CabinRemote cabin_1 = home.create(new Integer(1));
```

Мы создаем новый объект Cabin с помощью метода `create(Integer id)`, определенного в удаленном внутреннем интерфейсе компонента Cabin. Когда вызывается данный метод, внутренний объект взаимодействует с сервером EJB для того, чтобы создать компонент Cabin, добавив его данные в базу данных. Затем сервер EJB создает компонентный объект, заключающий в себе экземпляр компонента Cabin, и возвращает

клиенту удаленную ссылку на этот компонентный объект. После этого переменная cabin_1 будет содержать удаленную ссылку на компонент Cabin, только что нами созданный.

Для получения компонентного объекта из внутренней ссылки нам не нужен метод PortableRemoteObject.narrow(), поскольку она была объявлена как возвращающая тип CabinRemote. Не требуется никакого приведения. По той же причине нам не нужно явно сужать удаленные ссылки, возвращаемые методом findByPrimaryKey().

С помощью удаленной ссылки на компонентный объект мы можем модифицировать поля name, deckLevel, shipId и bedCount компонента Cabin:

```
CabinRemote cabin_1 = home.create(new Integer(1));
cabin_1.setName("Master Suite");
cabin_1.setDeckLevel(1);
cabin_1.setShipId(1);
cabin_1.setBedCount(3);
```

На рис. 4.4 показано, как должна выглядеть созданная нами таблица реляционной базы данных после выполнения этого кода. Она должна содержать одну запись.

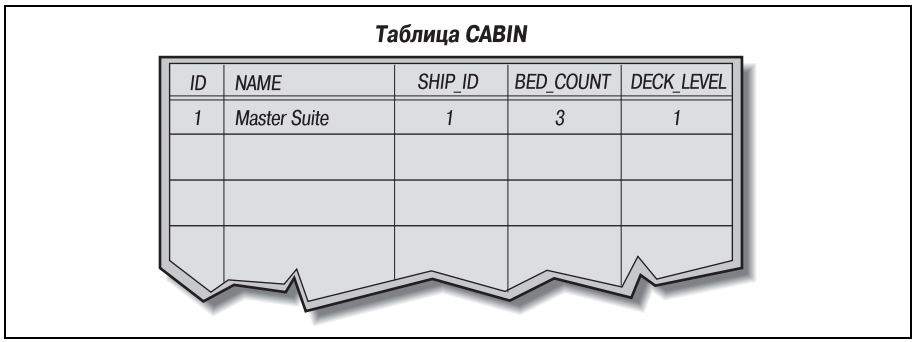

Рис. 4.4. Таблица CABIN с одной записью

Клиент может найти созданный объектный компонент посредством метода findByPrimaryKey() внутреннего интерфейса. Сначала мы создаем первичный ключ требуемого типа (в нашем случае – типа Integer). Когда мы вызываем поисковый метод внутреннего интерфейса, используя этот первичный ключ, нам возвращается удаленная ссылка на компонентный объект. Мы можем теперь запросить удаленную ссылку, возвращенную findByPrimaryKey(), и вернуть поля name, deckLevel, shipId и bedCount компонента Cabin:

```
Integer pk = new Integer(1);

CabinRemote cabin_2 = home.findByPrimaryKey(pk);
System.out.println(cabin_2.getName());
System.out.println(cabin_2.getDeckLevel());
```

```
System.out.println(cabin_2.getShipId());
System.out.println(cabin_2.getBedCount());
```

Теперь все готово к созданию и выполнению приложения `Client_1` для компонента Cabin, развернутого нами ранее. Откомпилируйте клиентское приложение и разверните компонент Cabin в контейнерной системе. Затем запустите приложение `Client_1`.

📖 Рабочее упражнение 4.1. Простой объектный компонент

После запуска приложения `Client_1` результат должен выглядеть примерно так:

```
Master Suite
1
1
3
```

Поздравляем! Вы только что создали и использовали ваш первый объектный компонент. Конечно, клиентское приложение делает очень немного. Прежде чем заняться разработкой сеансовых компонентов, создайте еще один клиент, который будет добавлять в базу данных некоторые тестовые данные. Здесь мы создадим `Client_2` как модификацию `Client_1`, которая заполняет базу данных большим количеством кают в трех различных судах:

```
package com.titan.cabin;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class Client_2 {

    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();

            Object ref = jndiContext.lookup("CabinHomeRemote");
            CabinHomeRemote home = (CabinHomeRemote)
                PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
            // Добавляем 9 кают к палубе 1 судна 1
            makeCabins(home, 2, 10, 1, 1);
            // Добавляем 10 кают к палубе 2 судна 1
            makeCabins(home, 11, 20, 2, 1);
```

```
            // Добавляем 10 кают к палубе 2 судна 1
            makeCabins(home, 21, 30, 3, 1);

            // Добавляем 10 кают к палубе 1 судна 2
            makeCabins(home, 31, 40, 1, 2);
            // Добавляем 10 кают к палубе 2 судна 2
            makeCabins(home, 41, 50, 2, 2);
            // Добавляем 10 кают к палубе 3 судна 2
            makeCabins(home, 51, 60, 3, 2);

            // Добавляем 10 кают к палубе 1 судна 3
            makeCabins(home, 61, 70, 1, 3);
            // Добавляем 10 кают к палубе 2 судна 3
            makeCabins(home, 71, 80, 2, 3);
            // Добавляем 10 кают к палубе 3 судна 3
            makeCabins(home, 81, 90, 3, 3);
            // Добавляем 10 кают к палубе 4 судна 3
            makeCabins(home, 91, 100, 4, 3);

            for (int i = 1; i <= 100; i++){
                Integer pk = new Integer(i);
                CabinRemote cabin = home.findByPrimaryKey(pk);
                System.out.println("PK = "+i+", Ship = "+cabin.getShipId()
                    + ", Deck = "+cabin.getDeckLevel()
                    + ", BedCount = "+cabin.getBedCount()
                    + ", Name = "+cabin.getName());
            }

        } catch (java.rmi.RemoteException re) {re.printStackTrace();}
          catch (javax.naming.NamingException ne) {ne.printStackTrace();}
          catch (javax.ejb.CreateException ce) {ce.printStackTrace();}
          catch (javax.ejb.FinderException fe) {fe.printStackTrace();}
    }

    public static javax.naming.Context getInitialContext()
        throws javax.naming.NamingException{
    Properties p = new Properties();
    // ... задаем свойства JNDI, специфичные для производителя
    return new javax.naming.InitialContext(p);
    }

    public static void makeCabins(CabinHomeRemote home, int fromId,
                                  int toId, int deckLevel, int shipNumber)
        throws RemoteException, CreateException {

    int bc = 3;
    for (int i = fromId; i <= toId; i++) {
        CabinRemote cabin = home.create(new Integer(i));
        int suiteNumber = deckLevel*100+(i-fromId);
        cabin.setName("Suite "+suiteNumber);
        cabin.setDeckLevel(deckLevel);
        bc = (bc==3)?2:3;
```

```
            cabin.setBedCount(bc);
            cabin.setShipId(shipNumber);
            }
        }
    }
```

Создайте и запустите приложение `Client_2` для компонента Cabin, развернутого вами ранее. `Client_2` генерирует длинный вывод, в котором перечислены все новые компоненты Cabin, только что добавленные в базу данных:

```
PK = 1, Ship = 1, Deck = 1, BedCount = 3, Name = Master Suite
PK = 2, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 100
PK = 3, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 101
PK = 4, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 102
PK = 5, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 103
PK = 6, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 104
PK = 7, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 105
...
```

Теперь таблица `CABIN` содержит 100 записей, представляющих 100 кают в вашей системе EJB. Это неплохой набор тестовых данных для сеансового компонента, который мы создадим в следующем разделе, и для всех последующих примеров этой книги.

Разработка сеансового компонента

Сеансовые компоненты действуют в качестве посредников клиента, управляя рабочим потоком (прикладным процессом) и заполняя пробелы, существующие между представлением данных объектными компонентами и прикладной логикой, взаимодействующей с этими данными. Сеансовые компоненты часто применяются для управления взаимодействием между объектными компонентами и могут выполнять сложные манипуляции компонентами при выполнении некоторых задач. Поскольку пока мы определили только один объектный компонент, то сосредоточимся на сложной манипуляции компонентом Cabin, а не на взаимодействиях между компонентом Cabin и другими объектными компонентами. Взаимодействия между объектными компонентами, происходящие внутри сеансовых компонентов, будут более детально изучены в главе 12.

Клиентские приложения и другие компоненты используют компонент Cabin различными способами. Некоторые из этих применений учитывались при создании компонента Cabin, а многие нет. В конце концов, объектный компонент представляет данные, в нашем случае – данные, описывающие каюту. Применения, для которых мы предоставляем эти данные, будут все время меняться. Следовательно, важным является само разделение данных и рабочего потока. В прикладной системе «Титан», например, нам может потребоваться создать список или

отчет по каютам такими способами, которые не были учтены при создании компонента Cabin. Вместо того, чтобы изменять компонент Cabin всякий раз, когда нам захочется взглянуть на него с другой стороны, мы будем получать требуемую нам информацию с помощью сеансового компонента. Изменение определения объектного компонента следует выполнять только в контексте более крупного процесса, например полного перепроектирования прикладной системы.

В главах 1 и 2 мы говорили о гипотетическом компоненте TravelAgent, отвечающем за рабочий поток, выполняющий регистрацию пассажира на круиз. Этот сеансовый компонент будет использоваться в клиентских приложениях, доступных туристическим агентствам по всему миру. Кроме резервирования, компонент TravelAgent предоставляет информацию о том, какие каюты в круизе свободны. В этой главе мы разработаем в компоненте TravelAgent первую реализацию такого списка. Метод, формирующий список, который мы разработаем в этом примере, по общему признанию довольно груб и далек от оптимального. Однако этот пример полезен для демонстрации того, как можно разработать очень простой сеансовый компонент без состояния, и как такие сеансовые компоненты могут управлять другими компонентами. В главе 12 мы перепишем этот метод. Реализация получения списка кают, разработанная здесь, будет использоваться туристическим агентом для предоставления клиентам списка кают, удовлетворяющих их запросам. Компонент Cabin не поддерживает этот вид списка непосредственно, он и не должен это делать. Список, который нам нужен, является специфичным для компонента TravelAgent, поэтому на него и возлагается ответственность за выполнение запроса компонента Cabin и формирование списка.

Вам понадобится создать для компонента TravelAgent рабочий каталог, как мы это сделали для компонента Cabin. Назовем этот каталог *travelagent* и разместим его ниже каталога */dev/com/titan*, содержащего также каталог *cabin* (рис. 4.5).

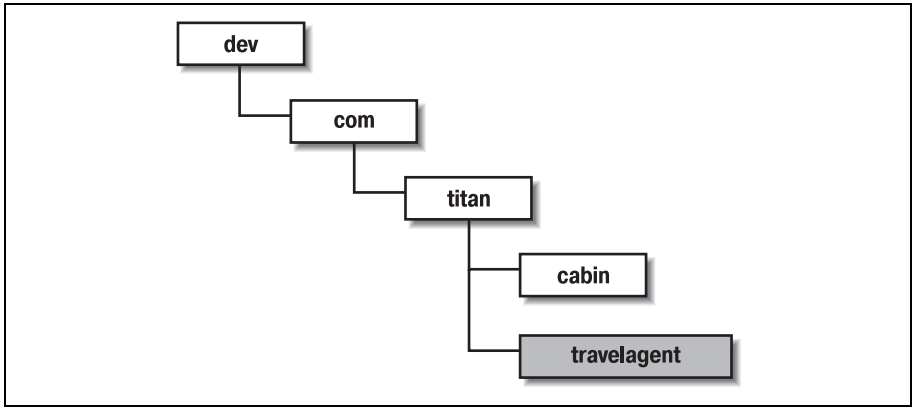

Рис. 4.5. Структура каталогов для компонента TravelAgent

Вы будете размещать все файлы Java и XML-дескриптор развертывания для компонента TravelAgent в этом каталоге.

TravelAgentRemote: Удаленный интерфейс

Как и прежде, начнем с определения удаленного интерфейса. Это позволит нам сосредоточить внимание на прикладной цели компонента, а не на его реализации. Начнем с начала. Мы знаем, что компонент TravelAgent должен будет предоставить метод для выдачи списка всех свободных кают с указанным количеством коек на указанном судне. Назовем этот метод `listCabins()`. Нам нужен только список имен кают и уровней палуб, поэтому мы определим `listCabins()` так, чтобы он возвращал массив строк. Здесь показан удаленный интерфейс `TravelAgentRemote`:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    // Стоковые элементы используют формат "id, name, deck level"
    public String [] listCabins(int shipID, int bedCount)
        throws RemoteException;
}
```

TravelAgentHomeRemote: Удаленный внутренний интерфейс

На втором шаге процесса разработки компонента TravelAgent нам необходимо создать удаленный внутренний интерфейс. В удаленном внутреннем интерфейсе сеансового компонента описываются конструирующие методы, которые инициализируют новый сеансовый компонент перед его использованием клиентом.

Поисковые методы в сеансовых компонентах не применяются, но применяются с объектными компонентами для поиска постоянных объектов с целью использования их на стороне клиента. В отличие от объектных компонентов сеансовые компоненты не постоянны и не представляют какие-либо данные в базе данных, поэтому поисковый метод не был бы нужен. Не существует никакой отдельной сессии, которую требовалось бы искать. Сеансовый компонент выделен для клиента на все время жизни этого клиента (или меньше). По той же причине мы не должны беспокоиться о первичных ключах: поскольку сеансовые компоненты не представляют постоянные данные, нам не нужен ключ для доступа к этим данным.

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
```

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {
    public TravelAgentRemote create()
        throws RemoteException, CreateException;
}
```

В случае с компонентом TravelAgent нам нужен только простой метод `create()` **для получения ссылки на компонент. Вызов этого метода** `create()` **возвращает удаленную ссылку компонента TravelAgent, с которой клиент может работать в процессе резервирования.**

TravelAgentBean: Класс компонента

Используя удаленный интерфейс в качестве ориентира, мы можем определить класс `TravelAgentBean`**, реализующий метод** `listCabins()`**. Следующий код содержит законченное определение** `TravelAgentBean` **для этого примера:**

```
package com.titan.travelagent;

import com.titan.cabin.CabinRemote;
import com.titan.cabin.CabinHomeRemote;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.Context;
import java.util.Properties;
import java.util.Vector;
import javax.rmi.PortableRemoteObject;
import javax.ejb.EJBException;


public class TravelAgentBean implements javax.ejb.SessionBean {

    public void ejbCreate() {
    // Ничего не делает
    }
    public String [] listCabins(int shipID, int bedCount) {

        try {
            javax.naming.Context jndiContext = new InitialContext();
            Object obj = jndiContext.lookup("java:comp/env/ejb/
                                             CabinHomeRemote");

            CabinHomeRemote home = (CabinHomeRemote)
                PortableRemoteObject.narrow(obj,CabinHomeRemote.class);

            Vector vect = new Vector();
            for (int i = 1; ; i++) {
                Integer pk = new Integer(i);
                CabinRemote cabin;
                try {
                    cabin = home.findByPrimaryKey(pk);
                } catch(javax.ejb.FinderException fe) {
                    break;
```

```
                }
                // Проверка на совпадение числа коек и идентификаторов судов
                if (cabin.getShipId() == shipID &&
                    cabin.getBedCount() == bedCount) {
                    String details =
                     = i+","+cabin.getName()+","+cabin.getDeckLevel();
                    vect.addElement(details);
                }
            }

            String [] list = new String[vect.size()];
            vect.copyInto(list);
            return list;

        } catch(Exception e) {throw new EJBException(e);}
    }

    private javax.naming.Context getInitialContext()
        throws javax.naming.NamingException {
        Properties p = new Properties();
        // ... задание свойств JNDI для специфичного производителя
        return new javax.naming.InitialContext(p);
    }

    public void ejbRemove(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void setSessionContext(javax.ejb.SessionContext cntx){}
}
```

Внимательно изучив метод `listCabins()`**, мы можем разложить его реализацию на части, начав с использования JNDI для поиска** `CabinHomeRemote`**:**

```
javax.naming.Context jndiContext = new InitialContext();

Object obj = jndiContext.lookup("java:comp/env/ejb/CabinHomeRemote");

CabinHomeRemote home = (CabinHomeRemote)
    javax.rmi.PortableRemoteObject.narrow(obj, CabinHomeRemote.class);
```

Компоненты являются клиентами других компонентов точно так же, как и клиентские приложения. Это означает, что они должны взаимодействовать с другими компонентами таким же образом, каким взаимодействуют с компонентами и клиентские приложения. Для того чтобы один компонент мог найти и использовать другой компонент, он должен сначала найти и получить ссылку на внутренний объект компонента. Это выполняется с помощью JNDI точно таким же способом, которым мы использовали JNDI в разработанных ранее приложениях `Client_1` **и** `Client_2` **для получения ссылки на компонент Cabin.**

У всех компонентов есть контекст JNDI по умолчанию, называемый контекстом имен окружения, рассмотренный кратко в главе 3. Кон-

текст по умолчанию находится в пространстве имен (каталоге) с именем "java:comp/env" и его подкаталогах. Во время развертывания компонента все компоненты, которые он использует, связываются с подкаталогом "java:comp/env/ejb" так, чтобы компонентные ссылки могли быть получены во время выполнения через простое и логичное использование контекста JNDI по умолчанию. Мы еще ненадолго вернемся к этому, когда будем рассматривать дескриптор развертывания для компонента TravelAgent.

Как вы узнали из главы 2, в EJB 2.0 компоненты могут иметь удаленные и/или локальные компонентные интерфейсы. Однако чтобы не усложнять этот первый ряд примеров, мы будем иметь дело только с удаленными компонентными интерфейсами. В главе 5 объясняется, как этот пример может быть реализован с помощью локальных интерфейсов.

Получив удаленный внутренний объект компонента Cabin, мы можем использовать его для формирования списка кают, которые соответствуют переданным параметрам. В следующем коде выполняется просмотр всех компонентов Cabin и формируется список, включающий каюты для указанного судна и с указанным количеством коек:

```
Vector vect = new Vector();
for (int i = 1; ; i++) {
    Integer pk = new Integer(i);
    CabinRemote cabin;
    try {
        cabin = home.findByPrimaryKey(pk);
    } catch(javax.ejb.FinderException fe){
        break;
    }
    // Check to see if the bed count and ship ID match.
    if (cabin.getShipId() == shipID && cabin.getBedCount() == bedCount) {
        String details = i+",'"+cabin.getName()+"',"+cabin.getDeckLevel();
        vect.addElement(details);
    }
}
```

Этот метод просто выполняет итерации по всем первичным ключам, получая удаленную ссылку на каждый компонент Cabin, существующий в системе, и проверяя, соответствуют ли его `shipId` и `bedCount` переданным параметрам. Цикл `for` продолжается до тех пор, пока не будет возбуждено исключение `FinderException`, которое, вероятно, произойдет, когда будет использоваться первичный ключ, не связанный с компонентом (возможно, что это не самый надежный код, но пока оставим его таким). Вслед за этим блоком кода мы просто копируем содержимое вектора (`Vector`) в массив и возвращаем это клиенту.

Хотя это довольно грубый подход к поиску требуемых компонентов Cabin (в главе 12 мы создадим метод получше), он адекватно отражает

наши текущие цели. Данный пример призван проиллюстрировать, что рабочий поток, связанный с данной реализацией списка, не включен в компонент Cabin и не встроен в клиентское приложение. Логика рабочего потока, является ли она процессом резервации или получения списка, помещается в сеансовый компонент.

Дескриптор развертывания компонента TravelAgent

В компоненте TravelAgent применяется XML-дескриптор развертывания, похожий на дескриптор для объектного компонента Cabin. Следующие разделы содержат файл *ejb-jar.xml*, предназначенный для развертывания компонента TravelAgent в EJB 2.0 и 1.1, соответственно. В главе 12 мы рассмотрим развертывание нескольких компонентов, находящихся в одном дескрипторе развертывания, но сейчас компоненты TravelAgent и Cabin разворачиваются отдельно.

EJB 2.0: Дескриптор развертывания

В EJB 2.0 дескриптор развертывания для компонента TravelAgent выглядит так:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <home>com.titan.travelagent.TravelAgentHomeRemote</home>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <ejb-ref>
                <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.cabin.CabinHomeRemote</home>
                <remote>com.titan.cabin.CabinRemote</remote>
            </ejb-ref>
            <security-identity><use-caller-identity/></security-identity>
        </session>
    </enterprise-beans>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: Дескриптор развертывания

В EJB 1.1 дескриптор развертывания для компонента TravelAgent выглядит так:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <home>com.titan.travelagent.TravelAgentHomeRemote</home>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <ejb-ref>
                <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.cabin.CabinHomeRemote</home>
                <remote>com.titan.cabin.CabinRemote</remote>
            </ejb-ref>
        </session>
    </enterprise-beans>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

EJB 2.0 и 1.1: Определение XML-элементов

Единственное существенное различие между дескрипторами развертывания для 2.0 и 1.1 – это имя DTD и добавочный элемент `<security-identity>`, в EJB 2.0 просто распространяющий идентификатор вызывающего объекта.

Все элементы, кроме `<session-type>` и `<ejb-ref>`, XML-дескриптора развертывания компонента TravelAgent должны быть вам знакомы: в нем используются многие из тех же элементов, что и для компонента Cabin. Элемент `<session-type>` может иметь значение либо `Stateful`, либо `Stateless`, указывая тип сеансового компонента. В этом случае мы определяем сеансовый компонент без состояния.

Элемент `<ejb-ref>` во время развертывания служит для связывания ссылки компонента, используемой внутри компонента TravelAgent. В этом случае элемент `<ejb-ref>` описывает компонент Cabin, который мы уже развернули. Элемент `<ejb-ref-name>` определяет имя, по которому компонент TravelAgent должен получать ссылку на внутренний объект Cabin. `<ejb-ref-type>` указывает контейнеру тип компонента, `Entity` или `Session`. Элементы `<home>` и `<remote>` определяют полные имена для внутреннего и удаленного интерфейсов компонента Cabin.

Во время развертывания компонента `<ejb-ref>` будет сопоставлен с компонентом Cabin, находящимся на сервере EJB. Этот процесс во многом зависит от применяемого продукта, но результат всегда должен быть одним и тем же. Компонент TravelAgent, выполнив поиск

через JNDI по контекстному имени "java:comp/env/ejb/CabinHomeRemote", получит удаленную ссылку на внутренний объект компонента Cabin. Назначение элемента <ejb-ref> состоит в том, чтобы устранить зависимость JNDI от конкретной сети и текущей реализации при получении удаленных компонентных ссылок. Это делает компонент более переносимым, поскольку положение провайдера услуг JNDI в сети может изменяться без влияния на код компонента и даже на XML-дескриптор развертывания.

Однако, как вы узнаете в главе 5, в EJB 2.0 при обращении компонентов друг к другу внутри одного сервера рекомендуется использовать вместо удаленных локальные ссылки. Локальные ссылки задаются с помощью элемента <ejb-local-ref>, который выглядит точно так же, как элемент <ejb-ref>.

Раздел дескриптора развертывания <assembly-descriptor> одинаков и для EJB 2.0, и для EJB 1.1:

```
<assembly-descriptor>
    <security-role>
        <description>
Эта роль представляет всех, кому разрешен полный доступ к Cabin EJB.
        </description>
        <role-name>everyone</role-name>
    </security-role>

    <method-permission>
        <role-name>everyone</role-name>
        <method>
            <ejb-name>TravelAgentEJB</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>

    <container-transaction>
        <method>
            <ejb-name>TravelAgentEJB</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Развертывание компонента TravelAgent

После создания XML-дескриптора развертывания вы готовы к размещению компонента TravelAgent в своем файле JAR и развертыванию его на сервере EJB.

Для того чтобы сделать компонент TravelAgent доступным клиентскому приложению, необходимо использовать утилиту развертывания

или мастер сервера EJB. Утилита развертывания считывает файл JAR для того, чтобы добавить компонент TravelAgent к окружению сервера EJB. Если сервер EJB не выдвигает никаких специальных требований, маловероятно, что понадобится изменять или добавлять к компоненту какие-либо новые атрибуты. В этом примере нам не потребуется создавать таблицу в базе данных, т. к. компонент TravelAgent использует компонент Cabin, а сам не является постоянным. Однако нам будет нужно сопоставить элемент `<ejb-ref>` дескриптора развертывания компонента TravelAgent с компонентом Cabin. Средство развертывания сервера EJB предоставит механизм для выполнения этого действия. Разверните компонент TravelAgent и переходите к следующему разделу.

Выполните тот же самый процесс по отношению к файлу JAR компонента TravelAgent, который использовался для компонента Cabin. Упакуйте класс компонента TravelAgent и его дескриптор развертывания в файл JAR и сохраните этот файл в каталоге *com/titan/travelagent*:

```
\dev % jar cf cabin.jar com/titan/travelagent/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\travelagent\*.class META-INF\
ejb-jar.xml
```

Возможно, сначала придется создать каталог *META-INF* и скопировать в него *ejb-jar.xml*. Компонент TravelAgent теперь завершен и готов для развертывания. Затем следует обратиться к инструментальным средствам вашего контейнера EJB для развертывания компонента TravelAgent в контейнерной системе.

Создание клиентского приложения

Чтобы убедиться, что наш сеансовый компонент работает, создадим простое клиентское приложение, использующее его. Этот клиент просто формирует список кают, связанных с судном 1 и имеющих число коек, равное 3. Его логика похожа на логику клиента, который мы создали ранее для проверки компонента Cabin: он создает контекст для поиска `TravelAgentHomeRemote`, создает компонент TravelAgent и вызывает метод `listCabins()` для формирования списка свободных кают. Далее приведен его код:

```
import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;
```

```
public class Client_3 {
    public static int SHIP_ID = 1;
    public static int BED_COUNT = 3;

    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();

            Object ref = jndiContext.lookup("TravelAgentHomeRemote");
            TravelAgentHomeRemote home = (TravelAgentHomeRemote)
               PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

            TravelAgentRemote travelAgent = home.create();

            // получение списка всех кают судна 1 с 1 койкой
            String list [] = travelAgent.listCabins(SHIP_ID,BED_COUNT);

            for(int i = 0; i < list.length; i++){
                System.out.println(list[i]);
            }

        } catch(java.rmi.RemoteException re){re.printStackTrace();}
            catch(Throwable t){t.printStackTrace();}
    }
    static public Context getInitialContext() throws Exception {
        Properties p = new Properties();
        // ... задание свойства JNDI, специфичных для производителя
        return new InitialContext(p);
    }
}
```

После успешного выполнения `Client_3` вывод должен выглядеть следующим образом:

```
1,Master Suite                ,1
3,Suite 101                   ,1
5,Suite 103                   ,1
7,Suite 105                   ,1
9,Suite 107                   ,1
12,Suite 201                   ,2
14,Suite 203                   ,2
16,Suite 205                   ,2
18,Suite 207                   ,2
20,Suite 209                   ,2
22,Suite 301                   ,3
24,Suite 303                   ,3
26,Suite 305                   ,3
28,Suite 307                   ,3
30,Suite 309                   ,3
```

Итак, мы благополучно завершили создание первой части сеансового компонента TravelAgent – метода, который получает список кают путем манипуляции компонентом Cabin.

📖 Рабочее упражнение 4.2. Простой сеансовый компонент

5

Клиентское представление

Разработка компонентов `Cabin` и `TravelAgent` должна была поднять вашу уверенность, но также могла породить множество вопросов. До сих пор мы опускали большую часть подробностей, связанных с разработкой, развертыванием и доступом к этим компонентам. В этой и следующих главах мы будем постепенно раскрывать все аспекты Enterprise JavaBeans, чтобы показать разработку приложений EJB во всех деталях.

В этой главе основное внимание уделяется объектным и сеансовым компонентам со стороны клиента. Компоненты, управляемые сообщениями, не рассматриваются в этой главе, они подробно обсуждаются в главе 13. Клиент в системе EJB, независимо от того, является ли он приложением или другим компонентом, не работает непосредственно с компонентами. Вместо этого клиенты взаимодействуют с набором интерфейсов, предоставляющих доступ к компонентам и их прикладной логике. Эти интерфейсы состоят из интерфейса JNDI и клиентского программного интерфейса EJB. Интерфейс JNDI позволяет нам находить компоненты и обращаться к ним независимо от их расположения в сети. Клиентский программный интерфейс EJB – набор интерфейсов и классов, используемых разработчиком на стороне клиента для взаимодействия с компонентами.

Наилучший подход при чтении этой главы состоит в том, чтобы изучить несколько особенностей клиентского представления и затем, придерживаясь примеров из рабочей книги, посмотреть все эти особенности в действии. Это даст вам практический опыт и намного более ясное понимание обсуждаемых понятий. Играйте, экспериментируйте и вскоре вы убедитесь, что усвоили основные принципы.

Поиск компонентов с помощью интерфейса JNDI

В главе 4 мы начинали клиентское приложение с создания объекта `InitialContext`, который затем был использован для получения удаленной ссылки на объекты Home компонентов Cabin и TravelAgent. `InitialContext` представляет собой часть более крупного программного интерфейса, называемого интерфейсом идентификации и маршрутизации (Java Naming and Directory Interface, JNDI). Мы организуем при помощи интерфейса JNDI поиск компонентов Home в сервере EJB точно так же, как вы могли бы организовать поиск номера домашнего телефона своего друга или делового партнера с помощью телефонной книги.

Интерфейс JNDI – это стандартный дополнительный пакет Java, представляющий собою единый программный интерфейс для доступа к широкому ряду услуг. В этом отношении он несколько напоминает интерфейс JDBC, также предоставляющий единый доступ к различным реляционным базам данных. Так же как JDBC позволяет создавать код, не заботясь о том, с какой базой данных выполняется соединение: с Oracle или Sybase, интерфейс JNDI позволяет писать код, способный обращаться к различным службам каталогов и системам имен, типа LDAP, Novell Netware NDS, Sun Solaris NIS +, службе имен CORBA и службам имен, предоставляемым серверами EJB. Для поддержки JNDI серверам EJB требуется организовать компоненты в структуру каталога и предоставить драйвер JNDI, называемый *провайдером услуг* (*service provider*), для доступа к этой структуре. Используя JNDI, предприятие может организовывать свои компоненты, услуги, данные и другие ресурсы в большую структуру виртуального каталога, которая может предоставлять мощный механизм связывания воедино разнородных систем.

Двумя самыми важными особенностями интерфейса JNDI являются его виртуальность и динамичность. Интерфейс JNDI виртуален, потому что он позволяет одной службе каталогов связываться с другой через простой URL. URL в JNDI аналогичны связям в HTML. Щелчок по связи в HTML позволяет пользователю загружать новую страницу из сети. Новая страница может быть загружена с того же самого компьютера, что и начальная страница, или с совершенно другого узла – положение связанной страницы прозрачно для пользователя. Аналогично с помощью JNDI вы можете спуститься вниз по каталогам к файлам, принтерам, внутренним объектам EJB и другим ресурсам посредством связей, подобных связям HTML. Каталоги и подкаталоги могут располагаться на том же компьютере или могут быть разнесены по нескольким местам. Пользователь не должен ни знать, ни заботиться о том, где эти каталоги расположены на самом деле. Как разработчик или администратор вы можете создать виртуальные каталоги, охватывающие ряд услуг, расположенных в различных местах.

Интерфейс JNDI динамический, потому что он позволяет драйверам JNDI (т. е. поставщикам услуг) загружать во время выполнения определенные типы служб каталогов. Драйвер связывает определенный вид службы каталогов со стандартными классами интерфейсов JNDI. Когда выбрана связь с отдельной службой каталогов, драйвер для этого типа службы автоматически загружается с узла, хранящего каталог, если он еще не находится на пользовательской машине. Автоматическая загрузка драйверов JNDI дает возможность клиенту просматривать доступные службы каталогов без необходимости знать заранее, какие виды услуг могут быть найдены.

JNDI позволяет клиенту приложения рассматривать EJB-сервер как набор каталогов, похожих на каталоги обычной файловой системы. После того как клиентское приложение найдет и получит с помощью JNDI удаленную ссылку на внутренний объект, клиент может использовать его для получения ссылки на компонентный объект прикладного компонента. В компонентах TravelAgent и Cabin, с которыми вы работали в главе 4, для получения объекта `InitialContext` интерфейса JNDI мы использовали метод `getInitialContext()`, который выглядел следующим образом:

```
public static Context getInitialContext()
    throws javax.naming.NamingException {
    Properties p = new Properties();
    // ... задаем свойства JNDI, специфичные для производителя
    return new javax.naming.InitialContext(p);
}
```

Начальный контекст (*initial context*) – это отправная точка для любого поиска в JNDI, он подобен корню файловой системы. Способ, которым создается начальный контекст, достаточно специфичен, но не особо сложен. Мы начинаем с таблицы свойств типа `Properties`. Она является по существу хеш-таблицей, в которую мы вносим различные значения, определяющие тип начального контекста, который необходимо получить.

Конечно, как было упомянуто в главе 4, этот код будет меняться в зависимости от того, как поставщик EJB реализовал JNDI. Например, в случае с сервером приложений Pramati метод `getInitialContext()` мог бы выглядеть примерно так:

```
public static Context getInitialContext() throws Exception {
    Hashtable p = new Hashtable();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.pramati.naming.client.PramatiClientContextFactory");
    p.put(Context.PROVIDER_URL, "rmi://127.0.0.1:9191");
    return new InitialContext(p);
}
```

Более детально JNDI рассмотрен в книге «Java™ Enterprise in a Nutshell» (Java™ Enterprise. Справочник) Дэвида Флэнагана (David Fla-

nagan), Джима Фарли (Jim Farley), Уильяма Кроуфорда (William Crawford) и Криса Магнуссона (Kris Magnusson), выпущенную издательством O'Reilly.

Удаленный клиентский интерфейс

От разработчиков компонентов требуется предоставить класс компонента, компонентные интерфейсы и для объектных компонентов – первичный ключ. Клиенту будут видны только компонентные интерфейсы и класс первичного ключа, а класс компонента – нет. Компонентные интерфейсы и первичный ключ в EJB взаимодействуют с клиентским программным интерфейсом.

В EJB версии 1.1 все клиенты независимо от того, находятся ли они в той же самой контейнерной системе или нет, должны работать с *удаленным клиентским интерфейсом* (*Remote Client API*). Это означает, что они во всех своих соединениях должны использовать удаленный интерфейс, удаленный внутренний интерфейс и систему Java RMI. В EJB 2.0 удаленные клиенты все еще должны применять удаленный клиентский интерфейс, но компоненты, расположенные в той же самой контейнерной системе, имеют возможность работать с *локальным клиентским интерфейсом* (*Local Client API*). Локальный клиентский интерфейс предоставляет локальные компонентные интерфейсы и свободен от ограничений и издержек удаленного клиентского интерфейса.

В этом разделе более подробно рассматриваются удаленные компонентные интерфейсы и первичный ключ, а также другие типы Java, образующие удаленный клиентский интерфейс, применяемый и в EJB 2.0 и в EJB 1.1. Это позволит вам лучше понимать, как используется удаленный клиентский интерфейс и как организованы его отношения с классом компонента на сервере EJB. В конце этой главы в разделе «EJB 2.0: Локальный клиентский интерфейс» мы рассмотрим применение локальных компонентных интерфейсов для классов чтения потоков EJB 2.0.

Java RMI–IIOP

Enterprise JavaBeans 2.0 и 1.1 определяет удаленные интерфейсы компонента в терминах протокола Java RMI-IIOP, обеспечивающего совместимость с CORBA. Это означает, что производитель может выбрать любой базовый протокол, в соответствии с которым удаленные клиенты будут получать доступ к компонентам, с условием, что он будет поддерживать типы интерфейсов и параметров, совместимые с Java RMI-IIOP. EJB 1.1 требовал только, чтобы канальный протокол, реализуемый поставщиками, поддерживал типы, совместимые с Java RMI-IIOP. Другими словами, типы интерфейсов и значения, фигурирующие в удаленных ссылках, должны быть совместимы с типами,

разрешенными в Java RMI-IIOP. Это гарантирует поддержку ранних реализаций Java RMI-IIOP и обеспечивает беспрепятственный переход других поставщиков, желающих применять в EJB 2.0 существующий протокол Java RMI-IIOP. В EJB 2.0 поставщики все еще могут предлагать другие протоколы, совместимые с Java RMI-IIOP, но в дополнение к любым собственным протоколам они также должны поддерживать протокол CORBA IIOP 1.2 в соответствии со спецификацией CORBA 2.3.1.

В интересах совместимости с типами Java RMI-IIOP поставщики EJB должны ограничить определение интерфейсов и параметров типами, соответствующими IIOP 1.2. Эти ограничения в действительности не так плохи, и вы, возможно, при разработке ваших компонентов их даже и не заметите, но важно знать, что они существуют. В нескольких следующих разделах обсуждается программная модель Java RMI-IIOP, применимая к обеим версиям EJB. Обратите внимание, что в EJB 2.0 локальные компонентные интерфейсы не являются интерфейсами Java RMI и не обязаны поддерживать IIOP 1.2 и использовать типы, совместимые с протоколом Java RMI-IIOP.

Типы возвращаемых значений, параметры и исключения Java RMI

Супертипы удаленного внутреннего интерфейса и удаленного интерфейса, `javax.ejb.EJBHome` и `javax.ejb.EJBObject`, расширяют интерфейс `java.rmi.Remote`. Ожидается, что как подтипы интерфейса Remote они будут соответствовать спецификации Java RMI для удаленных интерфейсов.

Типы возвращаемых значений и параметры. Удаленные компонентные интерфейсы должны следовать нескольким рекомендациям, некоторые из которых применяются к разрешенным типам возвращаемых значений и параметрам. Существуют два вида типов возвращаемых значений и параметров: *объявленные* (*declared*) типы, проверяемые транслятором, и *действительные* (*actual*) типы, проверяемые во время выполнения. Типы, которые могут применяться в Java RMI, – действительные типы. Для того чтобы быть совместимыми с Java RMI, эти типы, используемые в интерфейсах `java.rmi.Remote`, должны быть примитивами, типами `String`, `java.rmi.Remote` или сериализуемыми типами. Типы `Java.rmi.Remote` и сериализуемые типы не должны явно реализовать интерфейсы `java.rmi.Remote` и `java.io.Serializable`. Например, тип `java.util.Collection`, не расширяющий явно `java.io.Serializable`, представляет собой совершенно правильный тип для возвращаемых значений удаленного поискового метода при условии, что конкретный класс, реализующий `Collection`, – действительный тип, реализующий `java.io.Serializable`.

Java RMI не имеет никаких специальных правил относительно объявленных типов возвращаемых значений или типов параметра. Во время

выполнения считается, что тип, не являющийся типом `java.rmi.Remote`, – сериализуемый; в противном случае генерируется исключение. Действительный переданный тип не может быть проверен транслятором, он должен быть проверен во время выполнения.

Существует список типов, которые в Java RMI передаются в качестве параметров или возвращаются методом:

Примитивы

`byte`, `boolean`, `char`, `short`, `int`, `long`, `float`.

Сериализуемые типы Java

Любой класс, реализующий, или любой интерфейс, расширяющий `java.io.Serializable`.

Удаленные типы Java RMI

Любой класс, реализующий, или любой интерфейс, расширяющий `java.rmi.Remote`.

Сериализуемые объекты передаются в виде копии (т. е. по значению), а не по ссылке; это означает, что изменения в сериализуемом объекте на одном уровне не отразятся автоматически на других уровнях. Объекты, реализующие `Remote`, такие как `TravelAgentRemote` или `CabinRemote`, передаются в виде *удаленных ссылок* (*remote references*), и тут есть некоторое отличие от передачи сериализуемых объектов. Удаленная ссылка – это интерфейс `Remote`, реализованный заглушкой распределенного объекта. Удаленная ссылка, передаваемая в качестве параметра или возвращаемая из метода, является сериализованной заглушкой, переданной по значению, а не объектом, на который сервер удаленно ссылается при помощи заглушки. В главе 12 внутренний интерфейс компонента TravelAgent изменен таким образом, что метод `create()` принимает ссылку на компонент Customer в качестве своего единственного параметра:

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {
    public TravelAgentRemote create(CustomerRemote customer)
        throws RemoteException, CreateException;
}
```

Параметр `customer` представляет собой удаленную ссылку на компонент Customer, передаваемую в метод `create()`. Когда в Enterprise JavaBeans передается или возвращается удаленная ссылка, заглушка компонентного объекта передается в виде копии. Копия заглушки компонентного объекта указывает на тот же самый компонентный объект, что и первоначальная заглушка. Это приводит к тому, что и экземпляр компонента и клиент получают удаленные ссылки на один и тот же компонентный объект. Таким образом, изменения, сделанные клиентом, использующим удаленную ссылку, будут отражены и на экземпляре компонента, работающего с этой же самой удаленной

ссылкой. На рис. 5.1 и 5.2 показаны различия в Java RMI между сериализуемым объектом и параметром в виде удаленной ссылки.

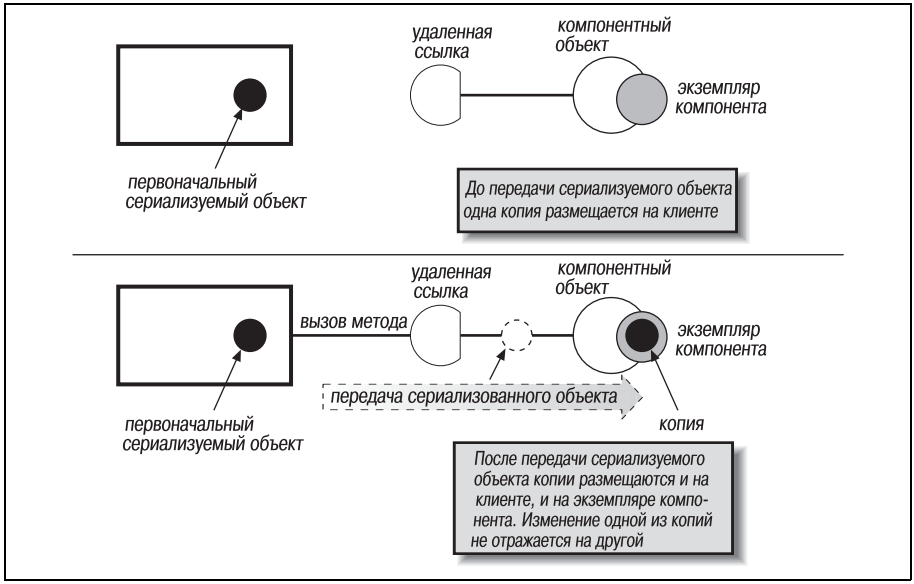

Рис. 5.1. Сериализуемые параметры в Java RMI

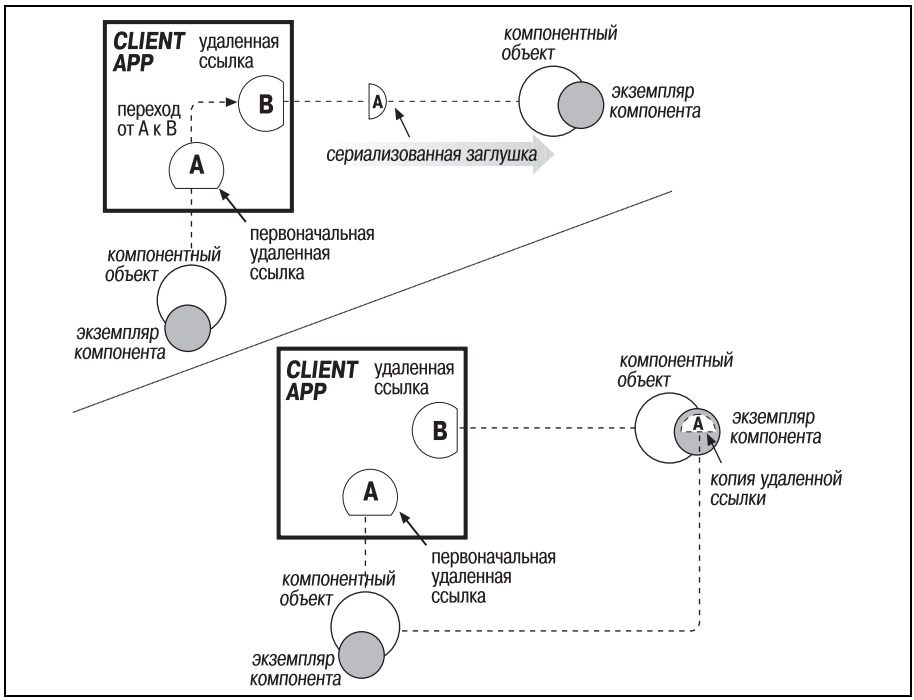

Рис. 5.2. Параметры в виде удаленной ссылки в Java RMI

Исключения. В спецификации Java RMI утверждается, что каждый метод, определенный в интерфейсе `Remote`, должен генерировать `java.rmi.RemoteException`. Исключение `RemoteException` используется, когда возникают проблемы соединения с распределенным объектом, такие как сетевой сбой или невозможность найти требуемый сервер. Кроме того, типы интерфейса `Remote` могут генерировать любые специфические для приложения исключения (определенные разработчиком приложения), которые могут потребоваться. Следующий код показывает удаленный интерфейс компонента `TravelAgent`, рассмотренного в главе 2. Этот удаленный интерфейс похож на тот, который мы определили в главе 4. `TravelAgentRemote` имеет несколько удаленных методов, включая `bookPassage()`. Метод `bookPassage()` может генерировать исключение `RemoteException` (как и требуется) и, дополнительно, прикладное исключение `IncompleteConversationalState`:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(int cruise)
        throws RemoteException, FinderException;
    public int getCruiseID() throws RemoteException;

    public void setCabinID(int cabin)
        throws RemoteException, FinderException;
    public int getCabinID() throws RemoteException;

    public int getCustomerID() throws RemoteException;

    public Ticket bookPassage(CreditCardRemote card, double price)
        throws RemoteException,IncompleteConversationalState;

    public String [] listAvailableCabins(int bedCount)
        throws RemoteException;

}
```

Ограничения типов в Java RMI–IIOP

В дополнение к программной модели Java RMI, обсужденной ранее, Java RMI-IIOP налагает дополнительные ограничения на удаленные интерфейсы и типы значений, применяемые в удаленном клиентском интерфейсе. Эти ограничения порождены ограничениями, наследуемыми из языка описания интерфейсов (Interface Definition Language – IDL), на котором основан протокол CORBA IIOP 1.2. Описание точного характера этих ограничений выходит за рамки данной книги. Здесь рассматриваются только два из них; другие, такие как пересечение имен IDL, встречаются настолько редко, что было бы неуместно даже упоминать их.[1]

[1] Желающих больше узнать о CORBA IDL и его отображении на язык Java адресуем к статьям «Common Object Request Broker: Architecture and Specification» (Простой посредник объектных запросов: архитектура и спецификация) и «The Java Language to IDL Mapping» (Отображение языка Java на IDL), которые можно найти на сайте OMG (*http://www.omg.org*).

- Перегрузка методов ограничена; удаленный интерфейс не может непосредственно расширять два или более интерфейса, имеющие методы с одинаковыми именами (даже если их параметры различны). Удаленный интерфейс может, однако, перегружать свои собственные методы и расширять удаленный интерфейс с перегруженными именами методов. Перегрузка просматривается здесь как включающая переопределение. На рис. 5.3 иллюстрируются обе эти ситуации.
- Сериализуемые типы не должны прямо или косвенно реализовать интерфейс `java.rmi.Remote`.

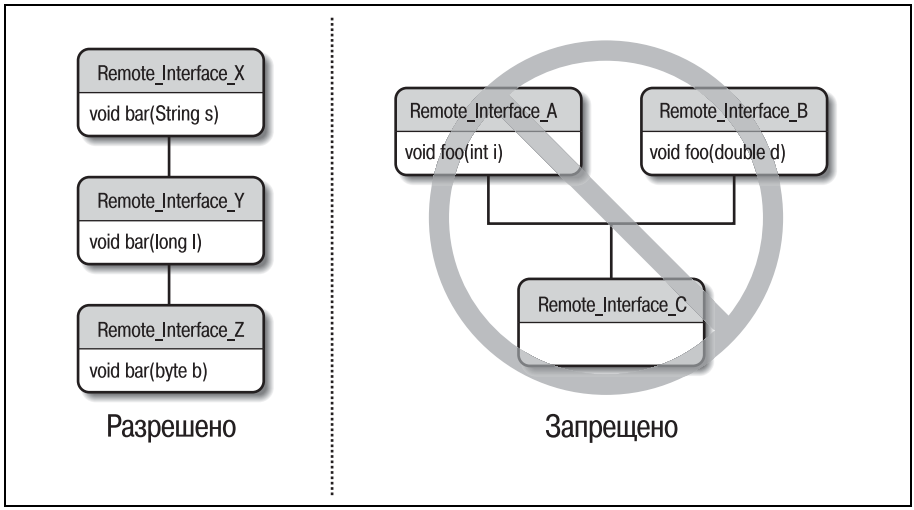

Рис. 5.3. Правила перегрузки при наследовании удаленных интерфейсов в Java RMI-IIOP

Явное сужение при помощи PortableRemoteObject

В Java RMI-IIOP удаленные ссылки должны быть явно сужены посредством метода `javax.rmi.PortableRemoteObject.narrow()`. Типичная для Java практика состоит в приведении ссылки к более определенному типу:

```
javax.naming.Context jndiContext;
...
CabinHomeRemote home =
(CabinHomeRemote)jndiContext.lookup("CabinHomeRemote");
```

Метод `javax.naming.Context.lookup()` возвращает `Object`. В EJB 2.0 в локальном клиентском интерфейсе мы можем допустить, что правомерно осуществлять приведение типа возвращаемого параметра. Однако удаленный клиентский интерфейс должен быть совместим с Java RMI-IIOP, а это означает, что клиенты должны придерживаться ограниче-

ний, налагаемых протоколом IIOP 1.2. Для того чтобы приспособиться ко всем языкам, многие из которых «не имеют никакого понятия» о приведении типов, IIOP 1.2 не поддерживает заглушки, реализующие множественные интерфейсы. Заглушка, возвращаемая в IIOP, реализует только интерфейс, определяемый типом возвращаемого значения вызываемого удаленного метода. Если типом возвращаемого значения является `Object`, как если бы удаленная ссылка возвращалась методом `lookup()`, заглушка будет реализовывать только методы, специфичные для типа `Object`.

Конечно, некоторые средства для преобразования удаленной ссылки от более общего к более определенному типу необходимы в объектно-ориентированной среде, поэтому Java RMI-IIOP предоставляет механизм для явного сужения ссылок до заданного типа. Метод `javax.rmi.PortableRemoteObject.narrow()` обобщает это сужение, для того чтобы предоставить сужение для IIOP, так же как и для других протоколов. Запомните, что, хотя удаленный клиентский интерфейс требует, чтобы использовались ссылки и типы аргументов Java RMI-IIOP, в качестве канального протокола не должен выступать IIOP 1.2. Другие протоколы кроме IIOP могут также требовать явного сужения. Класс `PortableRemoteObject` обобщает процесс сужения так, чтобы мог применяться любой протокол.

Для сужения возвращаемого параметра метода `Context.lookup()` к подходящему типу следует явно указать его удаленной ссылке, реализующей интерфейс:

```
import javax.rmi.PortableRemoteObject;
...
javax.naming.Context jndiContext;
...
Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

В случае успешного завершения метод `narrow()` возвращает заглушку, реализующую указанный удаленный интерфейс. Поскольку заглушка, как известно, реализует правильный тип, можно использовать «родное» приведение типов Java, чтобы сузить заглушку к правильному удаленному интерфейсу. Метод `narrow()` принимает два параметра: удаленную ссылку, которая должна быть сужена, и тип этого приведения. Определение метода `narrow()` следующее:[1]

```
package javax.rmi;

public class PortableRemoteObject extends java.lang.Object {
```

[1] Другие методы класса `PortableRemoteObject` не так важны для разработчика приложений EJB. Они предназначены для разработчиков Java RMI.

```
    public static java.lang.Object narrow(java.lang.Object narrowFrom,
        java.lang.Class narrowTo)
        throws java.lang.ClassCastException;
    ...
}
```

Метод `narrow()` должен применяться только в тех случаях, когда удаленная ссылка на компонент Home или компонент Object возвращена без указанного типа удаленного интерфейса. Это происходит в шести случаях:

1. **Удаленная ссылка на внутренний объект получена с помощью метода `javax.naming.Context.lookup()`:**

```
Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

2. **Удаленная ссылка на компонент Object получена из объектов `Collection` или `Enumeration`, возвращенных поисковым методом удаленного домашнего интерфейса:**

```
ShipHomeRemote shipHome = ... // получаем внутренную ссылку на ship
Enumeration enum = shipHome.findByCapacity(2000);
while(enum.hasMoreElements()){
    Object ref = enum.nextElement();
    ShipRemote ship = (ShipRemote)
        PortableRemoteObject.narrow(ref, ShipRemote.class);
    // используем ссылку на компонент Ship
}
```

3. **Удаленная ссылка на компонент Object получена с помощью метода `javax.ejb.Handle.getEJBObject()`:**

```
Handle handle = .... // получаем дескриптор
Object ref = handle.getEJBObject();
CabinRemote cabin = (CabinRemote)
PortableRemoteObject.narrow(ref,CabinRemote.class);
```

4. **Удаленная ссылка на компонент Home получена с помощью метода `javax.ejb.HomeHandle.GetEJBHome()`:**

```
HomeHandle homeHdle = ... // получаем внутренний дескриптор
EJBHome ref = homeHdle.getEJBHome();
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

5. **Удаленная ссылка на компонент Home получена с помощью метода `javax.ejb.EJBMetaData.GetEJBHome()`:**

```
EJBMetaData metaData = homeHdle.getEJBMetaData();
EJBHome ref = metaData.getEJBHome();
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

6. Удаленный компонент Object общего типа возвращается любым прикладным методом. Здесь приведен гипотетический пример:

```
// Officer расширяет Crewman
ShipRemote ship = // получаем удаленную ссылку на Ship
CrewmanRemote crew = ship.getCrewman("Burns", "John", "1st Lieutenant");
OfficerRemote burns = (OfficerRemote)
    PortableRemoteObject.narrow(crew, OfficerRemote.class);
```

Метод `PortableRemoteObject.narrow()` не требуется, если удаленный тип указан в сигнатуре метода. Это справедливо для создающих и поисковых методов в удаленных внутренних интерфейсах, возвращающих единственный компонент. Например, методы `create()` и `findByPrimaryKey()`, определенные в интерфейсе `CabinHomeRemote` (глава 4), не требуют применения метода `narrow()`, потому что эти методы возвращают уже правильный тип компонента Object. Прикладные методы, возвращающие правильный тип, также не должны использовать метод `narrow()`, что проиллюстрировано в следующем коде:

```
/* В методе CabinHomeRemote.create() в качестве
 * возвращаемого значения указан интерфейс CabinRemote,
 * поэтому явное сужение не требуется.*/
CabinRemote cabin = cabinHome.create(12345);

/* В методе CabinHomeRemote.findByPrimaryKey() в качестве
 * возвращаемого значения указан интерфейс CabinRemote,
 * поэтому явное сужение не требуется.*/
CabinRemote cabin = cabinHome.findByPrimaryKey(12345);

/* В прикладном методе ShipRemote.getCrewman() в качестве
 * возвращаемого значения указан интерфейс CrewmanRemote,
 * поэтому явное сужение не требуется.*/
CrewmanRemote crew = ship.getCrewman("Burns", "John", "1st Lieutenant");
```

Удаленный внутренний интерфейс

Удаленный внутренний интерфейс предоставляет действия жизненного цикла и метаданные компонента. Применяя JNDI для поиска компонента, мы получаем удаленную ссылку (или заглушку) на внутренний объект компонента, реализующего удаленный внутренний интерфейс. Каждый тип компонента может иметь один внутренний интерфейс, расширяющий интерфейс `javax.ejb.EJBHome`.

Здесь приведен интерфейс EJBHome:

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {
    public abstract EJBMetaData getEJBMetaData()
        throws RemoteException;
    public HomeHandle getHomeHandle()    // появился в 1.1
        throws RemoteException;
    public abstract void remove(Handle handle)
```

```
        throws RemoteException, RemoveException;
    public abstract void remove(Object primaryKey)
        throws RemoteException, RemoveException;
}
```

Удаление компонентов

Методы `EJBHome.remove()` отвечают за удаление компонента. Их параметры – это либо `javax.ejb.Handle` компонента, либо, для объектного компонента, его первичный ключ. Дескриптор (`Handle`) обсуждается более подробно позже, он по существу представляет собою сериализуемый указатель на указанный компонент. При вызове любого метода `EJBHome.remove()` удаленная ссылка компонента на стороне пользователя становится недействительной: заглушка компонента, который был удален, больше не действует. Если по некоторым причинам компонент не может быть удален, генерируется исключение `RemoveException`.

Воздействие метода `EJBHome.remove()` на сам компонент зависит от его типа. В случае с сеансовыми компонентами методы `EJBHome.remove()` прекращают обслуживание сеанса пользователя. Когда вызывается метод `EJBHome.remove()`, удаленная ссылка на сеансовый компонент становится недействительной, и все диалоговое состояние, поддерживаемое компонентом, теряется. Компонент TravelAgent, созданный нами в главе 3, не имеет состояния, поэтому в нем не существует никакого диалогового состояния (вы узнаете об этом больше в главе 12).

Когда метод `remove()` вызывается для объектных компонентов, удаленная ссылка становится недействительной, и любые данные, которые представляет компонент, действительно удаляются из базы данных. Это гораздо более разрушительное действие, т. к. если объектный компонент удален, данных, которые он представлял, больше не существует. Отличие в применении метода `remove()` для сеансового компонента и объектного компонента похоже на отличие между прекращением телефонного разговора и настоящим убийством абонента на другом конце линии. В обоих случаях разговор заканчивается, но несколько разными методами.

Следующий фрагмент кода взят из метода `main()` клиентского приложения, похожего на то, которые мы создали для проверки компонентов Cabin и TravelAgent. В нем показано, что можно удалять компоненты, используя первичный ключ (только для объектных компонентов) или дескриптор. Удаление объектного компонента уничтожает объект в базе данных, а удаление сеансового компонента приводит к тому, что удаленная ссылка на него становится недействительной. Далее приведен этот код:

```
Context jndiContext = getInitialContext();

// Получение списка всех кают с 3-мя койками на корабле 1

Object ref = jndiContext.lookup("TravelAgentHomeRemote");
```

```
TravelAgentHomeRemote agentHome = (TravelAgentHomeRemote)
    PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

TravelAgentRemote agent = agentHome.create();
String list [] = agent.listCabins(1,3);
System.out.println("Список 1: до удаления каюты с номером 30");
for(int i = 0; i < list.length; i++){
    System.out.println(list[i]);
}

// Получаем внутреннюю и удаленную каюту 30. Используем этот же список кают.

ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk = new Integer(30);
c_home.remove(pk);
list = agent.listCabins(1,3);
System.out.println("Список 2: после удаления каюты с номером 30");
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
}
```

Сначала приложение создает список кают, в том числе каюту с первичным ключом 30. Затем компонент Cabin с этим первичным ключом удаляется, и список создается заново. При повторном выполнении итерации каюта 30 будет пропущена. Метод `listCabin()` не сможет найти каюту с первичным ключом, равным 30, из-за того что компонент и его данные больше не существуют в базе данных. Результат должен быть таким:

```
Список 1: до удаления каюты с номером 30
1,Master Suite                  ,1
3,Suite 101                     ,1
5,Suite 103                     ,1
7,Suite 105                     ,1
9,Suite 107                     ,1
12,Suite 201                     ,2
14,Suite 203                     ,2
16,Suite 205                     ,2
18,Suite 207                     ,2
20,Suite 209                     ,2
22,Suite 301                     ,3
24,Suite 303                     ,3
26,Suite 305                     ,3
28,Suite 307                     ,3
29,Suite 309                     ,3
30,Suite 309                     ,3
Список 2: после удаления каюты с номером 30
1,Master Suite                  ,1
3,Suite 101                     ,1
```

```
5,Suite 103                    ,1
7,Suite 105                    ,1
9,Suite 107                    ,1
12,Suite 201                    ,2
14,Suite 203                    ,2
16,Suite 205                    ,2
18,Suite 207                    ,2
20,Suite 209                    ,2
22,Suite 301                    ,3
24,Suite 303                    ,3
26,Suite 305                    ,3
28,Suite 307                    ,3
29,Suite 308                    ,3
```

Метаданные компонента

Метод `EJBHome.getEJBMetaData()` возвращает экземпляр объекта `javax.ejb.EJBMetaData`, описывающего классы удаленного внутреннего интерфейса, удаленного интерфейса и первичного ключа, и указывает, является ли компонент сеансовым или объектным компонентом.[1] Этот тип метаданных представляет ценность для инструментальных средств Java, таких как интегрированные среды разработки, реализующих мастера и другие механизмы взаимодействия с компонентом с точки зрения клиента. Инструмент мог бы, например, использовать описание класса, предоставленное `EJBMetaData`, и механизм отражения Java для создания среды, в которой разработчики могут соединить все разворачиваемые компоненты. Конечно, также потребуется дополнительная информация, в частности имена JNDI- и URL-компонентов.

Большинство разработчиков приложений редко используют класс `EJBMetaData`. Однако знать о том, что он существует, полезно, если требуется создать автоматический генератор кода или другое автоматическое средство. В этих случаях знание интерфейса отражения необходимо.[2] Следующий код иллюстрирует определение интерфейса для `EJBMetaData`. Любой класс, реализующий интерфейс `EJBMetaData`, должен быть сериализуемым и не может являться заглушкой распределенного объекта. Это позволяет интегрированным средам разработки и другим инструментальным средствам сохранять `EJBMetaData` для дальнейшего использования:

```
public interface javax.ejb.EJBMetaData {
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
```

[1] У компонентов, управляемых сообщениями, в EJB 2.0 нет компонентных интерфейсов и к ним нельзя обратиться с помощью Java RMI-IIOP.

[2] Рассмотрение интерфейса отражения выходит за рамки этой книги, но он освещен в книге «Java in a Nutshell» Дэвида Флэнагана (David Flanagan), выпущенной издательством O'Reilly.

```
    public abstract Class getPrimaryKeyClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
}
```

Следующий код иллюстрирует, как с помощью класса EJBMetaData можно получить дополнительную информацию о компоненте Cabin. Обратите внимание, что не существует способа получить класс компонента с помощью EJBMetaData; класс компонента не входит в клиентский API и поэтому не является метаданными. Далее приведен этот код:

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

EJBMetaData meta = c_home.getEJBMetaData();

System.out.println(meta.getHomeInterfaceClass().getName());
System.out.println(meta.getRemoteInterfaceClass().getName());
System.out.println(meta.getPrimaryKeyClass().getName());
System.out.println(meta.isSession());
```

Это приложение выводит следующее:

```
com.titan.cabin.CabinHomeRemote
com.titan.cabin.CabinRemote
com.titan.cabin.CabinPK
false
```

Кроме предоставления типа компонентного класса, EJBMetaData дает возможность получить для компонента удаленный объект Home. Получив из EJBMetaData удаленный компонент Home, мы сможем получить ссылку на удаленный компонент Object, а также выполнять многие другие функции. В следующем коде мы при помощи класса EJBMetaData получаем из объекта Home класс первичного ключа, создаем экземпляр ключа, получаем удаленный компонент Home и удаленную ссылку на компонент Object для указанного компонента Cabin:

```
Object primKeyType = meta.getPrimaryKeyClass();
if(primKeyType instanceof java.lang.Integer){
    Integer pk = new Integer(1);

    Object ref = meta.getEJBHome();
    CabinHomeRemote c_home2 = (CabinHomeRemote)
        PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

    CabinRemote cabin = c_home2.findByPrimaryKey(pk);
    System.out.println(cabin.getName());
}
```

HomeHandle

HomeHandle можно получить, вызвав метод EJBHome.getHomeHandle(). Этот метод возвращает объект javax.ejb.HomeHandle, предоставляющий сериализуемую ссылку на удаленный (remote) компонент Home. HomeHandle позволяет сохранить эту удаленную (remote) внутреннюю ссылку с тем, чтобы ее можно было использовать позже. Она похожа на javax.ejb.Handle и ниже обсуждается более подробно.

Создание и поиск компонентов

В дополнение к стандартному методу javax.ejb.EJBHome, который наследуют все удаленные внутренние интерфейсы, данные интерфейсы включают также специальные *конструирующие* и *поисковые* методы. Мы уже говорили о конструирующих и поисковых методах, но небольшой обзор укрепит ваше понимание роли удаленного внутреннего интерфейса в удаленном клиентском API. Следующий код иллюстрирует удаленный внутренний интерфейс, определенный для компонента Cabin:

```
public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;

    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

Конструирующие методы генерируют CreateException, если в процессе создания что-то происходит не так, как надо. Поисковые методы в случае ошибки генерируют FinderException. Поскольку данные методы определены в интерфейсе, наследуемом от Remote, они также должны объявить, что они генерируют RemoteException.

Конструирующие и поисковые методы специфичны для каждого компонента, поэтому определение подходящих конструирующих и поисковых методов в удаленном внутреннем интерфейсе оставлено на усмотрение разработчика. CabinHomeRemote в настоящее время содержит только один конструирующий метод, который создает каюту с указанным идентификатором (ID), и один поисковый метод, выполняющий поиск компонента по его первичному ключу. Однако нетрудно придумать методы, создающие и находящие каюту с заданными свойствами, например, каюту с тремя ложами или салон-люкс с голубыми обоями.

Поисковые методы есть только у объектных компонентов; у сеансовых компонентов их нет. Объектные компоненты представляют собой уникальные идентифицируемые данные внутри базы данных и поэтому могут быть найдены. Сеансовые компоненты, с другой стороны, не представляют данные: они создаются для обслуживания клиентских приложений и не имеют постоянства, поэтому в них нечего искать. Поисковый метод для сеансового компонента был бы бессмыслен.

В EJB 2.0 конструирующие методы были расширены таким образом, что имя метода стало возможно использовать в качестве суффикса. Другими словами, все конструирующие методы могут принимать форму `create<СУФФИКС>()`. Например, для компонента Customer можно было бы определить удаленный внутренний интерфейс с несколькими конструирующими методами, каждый из которых принимает свой параметр типа `String`, но имеет различные имена методов:

```
public interface CustomerHome extends javax.ejb.EJBHome {

    public CustomerRemote createWithSSN(Integer id, String
socialSecurityNumber)
        throws CreateException, RemoteException;

    public CustomerRemote createWithPIN(Integer personalIdNumber)
        throws CreateException, RemoteException;

    public CustomerRemote createWithBLN(Integer id, String
businessLicenseNumber)
        throws CreateException, RemoteException;

    public Customer findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;
 }
```

Хотя использование суффиксов в именах конструирующих методов в EJB 2.0 разрешено, это не является обязательным. EJB 1.1 не поддерживает использование суффиксов в именах конструирующих методов.

Конструирующие и поисковые методы, определенные в удаленных внутренних интерфейсах, являются прямыми и могут без труда использоваться клиентом. Конструирующие методы внутреннего интерфейса должны соответствовать методам `ejbCreate()` и `ejbPostCreate()` класса компонента. Методы `create()`, `ejbCreate()` и `ejbPostCreate()` соотносятся, только если они имеют одинаковые параметры, если эти параметры имеют одинаковые типы и следуют в одном и том же порядке и если их имена методов совпадают. Этим способом, когда клиент вызывает конструирующий метод внутреннего интерфейса, запрос может быть перенаправлен соответствующим методам `ejbCreate()` и `ejbPostCreate()` экземпляра компонента.

Поисковые методы во внутреннем интерфейсе работают так же, как и в компонентах, реализующих постоянство, в EJB 2.0 и 1.1. Каждый метод `find<СУФФИКС>()` внутреннего интерфейса должен соответствовать методу `ejbFind<СУФФИКС>()` самого компонента. Компоненты, управляемые контейнером, не реализуют методы `ejbFind()` в классе компонента, контейнер автоматически поддерживает поисковые методы. О том, как реализуются методы `ebjCreate()`, `ejbPostCreate()` и `ejbFind()` компонента, можно больше узнать в главах с 6 по 11.

EJB 2.0: Внутренние методы

Кроме поисковых и конструирующих методов в EJB 2.0 во внутреннем интерфейсе объектного компонента могут также быть определены *внутренние методы* (*home methods*). Внутренний метод – это прикладной метод; он может быть вызван из внутреннего интерфейса (локального или удаленного) и не специфичен для одного экземпляра компонента. Например, в компоненте Cabin может быть определен внутренний метод, возвращающий число кают на указанном уровне палуб:

```
public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;

    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;

    public int getDeckCount(int level) throws RemoteException;
}
```

Любой метод, определенный во внутреннем интерфейсе и не относящийся к конструирующим и поисковым, считается внутренним методом и должен иметь соответствующий метод `ejbHome()` в классе компонента, как показано ниже:

```
public class CabinBean implements javax.ejb.EntityBean{
    public int ejbHomeGetDeckCount(int level){
        // реализуем логику нахождения количества палуб
    }
    ...
}
```

Клиенты могут использовать внутренние методы из внутреннего интерфейса компонента. Клиенту не нужно получать ссылку на отдельный компонент Object:

```
Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

int count = home.getDeckCount(2);
```

Доступ к внутренним методам открыт только для объектных компонентов, а не для сеансовых. Они могут применяться для универсальной прикладной логики, связанной с изменением группы объектных компонентов, или получением информации, не специфичной для отдельного объектного компонента. Внутренние методы обсуждаются более подробно в главе 11.

Удаленный интерфейс

Прикладные методы компонента могут быть определены в удаленном интерфейсе, предоставляемом разработчиком компонента. Интерфейс `javax.ejb.EJBObject`, расширяющий интерфейс `java.rmi.Remote`, является базовым классом для всех удаленных интерфейсов.

Следующий код определяет удаленный интерфейс для компонента TravelAgent, созданный нами в главе 4:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public String [] listCabins(int shipID, int bedCount)
        throws RemoteException;
}
```

На рис. 5.4 показана иерархия наследования интерфейса `TravelAgentRemote`.

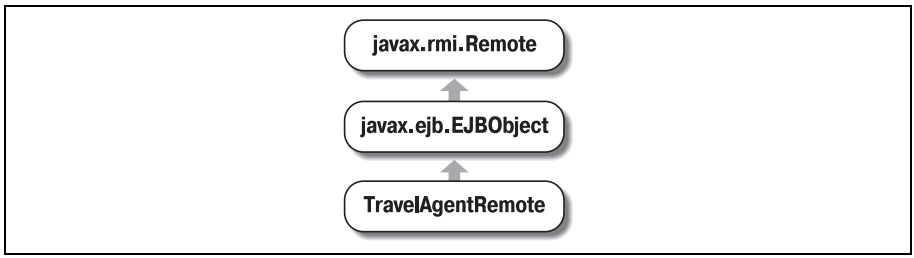

Рис. 5.4. Иерархия наследования интерфейсов компонента

Удаленные интерфейсы направлены на решение прикладной задачи и не включают методы для операций системного уровня, таких как постоянство, безопасность, параллелизм или обработка транзакций. Операции системного уровня обрабатываются сервером EJB, освобождающим клиента-разработчика от многих забот. Все методы удаленного интерфейса компонентов должны генерировать, по крайней мере, исключение `java.rmi.RemoteException`, указывающее на проблемы в распределенных коммуникациях. Кроме этого, методы в удаленном интерфейсе могут генерировать столько пользовательских исключений, сколько потребуется для того, чтобы указать на неверные ситуации в прикладной задаче или ошибки при выполнении прикладного метода. Подробности об определении пользовательских исключений можно найти в главах 12 и 14.

📖 Рабочее упражнение 5.1. Удаленные компонентные интерфейсы

Интерфейс EJBObject, дескриптор и первичный ключ

Все удаленные интерфейсы расширяют интерфейс `javax.ejb.EJBObject`, который предоставляет набор сервисных методов и возвращаемых

ими типов. Эти методы и возвращаемые типы полезны для управления взаимодействием между клиентом и компонентами. Здесь приведено определение интерфейса EJBObject:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome()
        throws RemoteException;
    public abstract Handle getHandle()
        throws RemoteException;
    public abstract Object getPrimaryKey()
        throws RemoteException;
    public abstract boolean isIdentical(EJBObject obj)
        throws RemoteException;
    public abstract void remove()
        throws RemoteException, RemoveException;
}
```

Когда клиент получает ссылку на удаленный интерфейс (фактически он получает удаленную ссылку на компонентный объект). Компонентный объект реализует удаленный интерфейс, перенаправляя вызовы прикладных методов классу компонента. Он предоставляет свою собственную реализацию методов EJBObject. Эти методы возвращают информацию о соответствующем экземпляре компонента, находящегося на сервере. Как говорилось в главе 2, компонентный объект автоматически генерируется во время развертывания компонента на сервере EJB, поэтому разработчику компонента не требуется создавать реализацию EJBObject.

Получение EJBHome

Метод EJBObject.getEJBHome() возвращает удаленную ссылку на внутренний объект компонента. Удаленная ссылка возвращается в виде объекта javax.ejb.EJBHome, который может быть сужен к удаленному внутреннему интерфейсу, специфичному для данного компонента. Этот метод полезен в случаях, когда компонентный объект покидает область видимости создавшего его удаленного внутреннего объекта. Поскольку удаленные ссылки, как и любые объекты Java, можно передавать в виде ссылок и возвращать из методов, удаленная ссылка и соответствующий ей удаленный объект могут быстро оказаться в совершенно разных частях приложения. Следующий код является несколько надуманным, но он объясняет, как удаленная ссылка может выйти из области видимости ее объекта и как может использоваться getEJBHome() для получения новой ссылки на внутренний объект, когда это необходимо:

```
public static void main(String [] args) {
    try {
        Context jndiContext = getInitialContext();
        Object ref = jndiContext.lookup("TravelAgentHomeRemote");
```

```
        TravelAgentHomeRemote home = (TravelAgentHomeRemote)
            PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

        // Получаем удаленную ссылку на компонент (компонентный объект)
        TravelAgentRemote agent = home.create();
        // Передаем эту удаленную ссылку некоторому методу
        getTheEJBHome(agent);

        } catch (java.rmi.RemoteException re){re.printStackTrace();}
          catch (Throwable t){t.printStackTrace();}
}

public static void getTheEJBHome(TravelAgentRemote agent)
    throws RemoteException {

    // Внутренний интерфейс находится вне области видимости метода,
    // поэтому он должен быть получен из компонентного объекта
    Object ref = agent.getEJBHome();
    TravelAgentHomeRemote home = (TravelAgentHomeRemote)
        PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);
    // Делаем с внутренним интерфейсом что-нибудь полезное
}
```

Первичный ключ

Метод `EJBObject.getPrimaryKey()` возвращает первичный ключ объектного компонента. Этот метод поддерживается только компонентными объектами, представляющими объектные компоненты. Объектные компоненты представляют специфичные данные, которые могут быть идентифицированы по этому первичному ключу. Сеансовые компоненты представляют задачи или процессы, а не данные, поэтому для этого типа компонентов первичные ключи не имеют смысла. Чтобы лучше понять природу первичного ключа, мы должны посмотреть на уровень контейнера EJB, рассмотренного в главах 2 и 3, выйдя за рамки клиентского представления.

EJB-контейнер отвечает за постоянство объектных компонентов, но реализация конкретного механизма постоянства оставлена на усмотрение производителя. Для того чтобы найти экземпляр компонента в постоянной памяти, необходимо поставить данные, образующие объект, в соответствие некоторому уникальному ключу. В реляционных БД данные уникально идентифицируются значением одного или нескольких столбцов, которые могут быть объединены для формирования первичного ключа. В объектно-ориентированной базе данных ключ содержит идентификатор объекта (Object ID, OID) или указатель базы данных некоторого типа. Независимо от механизма, который действительно несущественен с точки зрения клиента, уникальный ключ для данных компонента включается в состав первичного ключа, возвращаемого методом `EJBObect.getPrimaryKey()`.

Первичный ключ может применяться для получения удаленных ссылок на объектные компоненты с помощью метода `findByPrimaryKey()`.

С точки зрения клиента объект первичного ключа может быть использован для идентификации уникального объектного компонента. Очень важно понять контекст уникальности первичного ключа, как показано в следующем коде:

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

CabinRemote cabin_1 = home.create(101);
Integer pk = (Integer)cabin_1.getPrimaryKey();
CabinRemote cabin_2 = home.findByPrimaryKey(pk);
```

В этом коде клиент создает компонент Cabin, получает его первичный ключ и затем с помощью этого ключа получает новую ссылку на тот же самый компонент Cabin. Таким образом, у нас есть две переменные – `cabin_1` и `cabin_2`, представляющие собой удаленные ссылки на компонентные объекты. Обе они ссылаются на один и тот же компонент Cabin, с одними и теми данными, т. к. они имеют одинаковый первичный ключ.

Первичный ключ должен применяться для правильного компонента в правильном контейнере. Хотя это кажется достаточно очевидным, отношения первичного ключа с определенным контейнером и внутренним интерфейсом довольно важны. Первичный ключ гарантирует возвращение одного и того же объекта только в случае, когда он используется в пределах породившего его контейнера. В качестве примера представьте, что третий производитель продает компонент Cabin в виде законченного продукта. Производитель продает компонент Cabin и «Титану», и его конкуренту. Обе компании развертывают объектный компонент, используя свои собственные реляционные базы данных со своими собственными данными. Первичный ключ типа `Integer` со значением 20 в системе «Титана» не будет соответствовать тому же самому объекту Cabin в системе конкурента. У обеих туристических компаний есть компонент Cabin с первичным ключом равным 20, но они соответствуют разным каютам на разных судах. Эти компоненты Cabin находятся в разных контейнерах EJB, поэтому их первичные ключи не эквивалентны.[1] Каждый компонентный объект уникален в пределах своего внутреннего объекта. Два компонентных объекта, имеющие один и тот же внутренний объект и один и тот же первичный ключ, считаются одинаковыми.

Первичный ключ должен реализовать интерфейс `java.io.Serializable`. Это означает то, что первичный ключ, независимо от его формы, мо-

[1] Это, конечно же, неверно, если оба компонента Cabin работают с одной базой данных, что достаточно типично в сценариях с использованием кластеров.

жет быть получен из компонентного объекта, сохранен на стороне клиента с помощью механизма сериализации Java и, в случае необходимости, десериализован. После чего первичный ключ может быть использован для получения удаленной ссылки на тот же самый объектный компонент с помощью `findByPrimaryKey()`, – при условии, что этот ключ применяется к правильному удаленному внутреннему интерфейсу и контейнеру. Сохранение первичного ключа с помощью сериализации может быть полезно, если позднее клиентскому приложению нужно будет обратиться к определенным объектным компонентам.

Следующий код показывает первичный ключ, который был сериализован, а затем десериализован для того, чтобы повторно получить удаленную ссылку на тот же самый компонент:

```
// Получаем каюту 101 и устанавливаем ее имя
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
CabinRemote cabin_1 = home.findByPrimaryKey(pk_1);
cabin_1.setName("Presidential Suite");

// Сериализуем первичный ключ каюты 101 в файл
FileOutputStream fos = new FileOutputStream("pk101.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(pk_1);
outStream.flush();
outStream.close();
pk_1 = null;

// Десериализуем первичный ключ для каюты 101.
FileInputStream fis = new FileInputStream("pk101.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
Integer pk_2 = (Integer)inStream.readObject();
inStream.close();

// Повтоно получаем удаленную ссылку на каюту 101 и считываем ее имя
CabinRemote cabin_2 = home.findByPrimaryKey(pk_2);
System.out.println(cabin_2.getName());
```

Проверка компонентов на идентичность

Метод `EJBObject.isIdentical()` сравнивает две удаленные ссылки на компонентный объект. Следует рассмотреть, почему `Object.equals()` недостаточен для сравнения компонентных объектов. Компонентный объект представляет собой заглушку для распределенного объекта и поэтому содержит много переменных, в том числе связанных с сетевыми коммуникациями. В результате ссылки на два компонентных объекта могут быть неравны, даже в том случае, если они обе представля-

ют один и тот же отдельный компонент. Метод `EJBObject.isIdentical()` возвращает true, если две ссылки на компонентный объект представляют один и тот же компонент, даже если заглушки компонентного объекта представляют собой разные объектные экземпляры.

В следующем фрагменте кода показано, как это работает. Начинается все с создания двух удаленных ссылок на компонент TravelAgent. Эти удаленные компонентные объекты ссылаются на один и тот же тип компонента. Сравнение их с помощью `isIdentical()` возвращает true. Эти два компонента TravelAgent были созданы отдельно, но из-за того что у них нет состояния, они считаются эквивалентными. Если бы компонент TravelAgent был компонентом с состоянием (которым он станет в главе 12), результат был бы другим. Сравнение этим способом двух компонентов с состоянием приведет к результату false, поскольку эти компоненты хранят состояние связи, делающее их уникальными. Применяя `CabinHomeRemote.findByPrimaryKey()` для поиска двух компонентных объектов, которые ссылаются на один и тот же объектный компонент Cabin, мы знаем, что эти компоненты идентичны, потому что мы использовали один и тот же первичный ключ. В этом случае `isIdentical()` также возвратит true, потому что обе удаленные ссылки компонентного объекта указывают на один и тот же объект. Вот этот код:

```
Context ctx  = getInitialContext();

Object ref = ctx.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote agentHome =(TravelAgentHomeRemote)
    PortableRemoteObject.narrow(ref, TravelAgentHomeRemote.class);

TravelAgentRemote agent_1 = agentHome.create();
TravelAgentRemote agent_2 = agentHome.create();
boolean x = agent_1.isIdentical(agent_2);
// x будет равен true; эти два компонентных объекта равны

ref = ctx.lookup("CabinHomeRemote");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
Integer pk_2 = new Integer(101);
CabinRemote cabin_1 = c_home.findByPrimaryKey(pk_1);
CabinRemote cabin_2 = c_home.findByPrimaryKey(pk_2);
x = cabin_1.isIdentical(cabin_2);
// x будет равен true; эти два компонентных объекта равны
```

Первичный ключ типа `Integer` из компонента Cabin достаточно прост. Для того чтобы метод `EJBObject.isIdentical()` мог работать с более сложными, пользовательскими первичными ключами, от нас требуется перегрузить методы `Object.equals()` и `Object.hashCode()`. В главе 11 обсуждается процесс создания более сложных первичных ключей, называемых составными первичными ключами.

Удаление компонентов

Метод EJBObject.Remove() предназначен для удаления сеансовых компонентов и объектных компонентов. Действует этот метод так же, как и метод EJBHome.remove(), рассмотренный выше. Для сеансовых компонентов вызов remove() приводит к освобождению сессии и делает удаленную ссылку на компонентный объект недействительной. В случае объектных компонентов существующие данные объекта удаляются из базы данных, а удаленная ссылка становится недействительной. Следующий код показывает применение метода EJBObject.remove():

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

Integer pk = new Integer(101);
CabinRemote cabin = c_home.findByPrimaryKey(pk);
cabin.remove();
```

Если по какой-либо причине ссылка не может быть удалена, метод remove() возбуждает исключение RemoveException.

Дескриптор компонента

Метод EJBObject.getHandle() возвращает объект javax.ejb.Handle. Дескриптор (handle) – сериализуемая ссылка на удаленный компонентный объект. Это значит, что клиент может сохранить объект Handle с помощью сериализации Java, а затем десериализовывать его для того, чтобы повторно получить ссылку на тот же самый удаленный компонентный объект. Это похоже на сериализацию и повторное использование первичного ключа. Дескриптор позволяет нам воссоздать удаленную ссылку на компонентный объект, указывающую на тот же самый тип сеансового компонента или тот же самый уникальный объектный компонент, от которого произошел данный дескриптор.

Далее приведено определение интерфейса дескриптора:

```
public interface javax.ejb.Handle {
    public abstract EJBObject getEJBObject()
      throws RemoteException;
}
```

В интерфейсе Handle определен только один метод – getEJBObject(). Вызов этого метода возвращает удаленный компонентный объект, создавший дескриптор. Получив объект, можно сузить его к нужному типу удаленного интерфейса. Следующий код показывает, как можно сериализовать и десериализовать дескриптор на стороне клиента:

```
// Получаем каюту 100
Context jndiContext = getInitialContext();
```

```
Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
CabinRemote cabin_1 = home.findByPrimaryKey(pk_1);

// Сериализуем дескриптор каюты 100 в файл
Handle handle = cabin_1.getHandle();
FileOutputStream fos = new FileOutputStream("handle100.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(handle);
outStream.flush();
fos.close();
handle = null;

// Десериализуем дескриптор каюты 100
FileInputStream fis = new FileInputStream("handle100.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
handle = (Handle)inStream.readObject();
fis.close();

// Повторно получаем удаленную ссылку на каюту 100 и считываем ее имя

ref = handle.getEJBObject();
CabinRemote cabin_2 = (CabinRemote)
    PortableRemoteObject.narrow(ref, CabinRemote.class);

if(cabin_1.isIdentical(cabin_2))
    // всегда true
```

На первый взгляд кажется, что дескриптор и первичный ключ выполняют одни и те же задачи, но на самом деле это не так – задачи совершенно разные. Для использования первичного ключа необходимо наличие корректного удаленного внутреннего объекта: если у нас больше нет ссылки на удаленный внутренний объект, то нам необходимо с помощью JNDI найти контейнер и получить новый внутренний объект. Только после этого мы сможем вызвать метод `findByPrimaryKey()` **для поиска настоящего компонента. Следующий код показывает, как это должно работать:**

```
// Получаем первичный ключ из входного потока
Integer primaryKey = (Integer)inStream.readObject();

// Используем JNDI API для получения корневого каталога, или начального
// контекста
javax.naming.Context ctx = new javax.naming.InitialContext();

// Используем начальный контекст для получения EJBHome для компонента Cabin

Object ref = ctx.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
```

```
// Получаем ссылку на компонентный объект, представляющий экземпляр компонента
CabinRemote cabin_2 = home.findByPrimaryKey(primaryKey);
```

С объектом `Handle` легче работать, потому что он содержит в себе все операции, связанные с выполнением поиска контейнера через JNDI. В случае применения дескриптора корректный компонентный объект может быть получен в результате одного запроса метода `Handle.getEJBObject()`, что не требует вызова трех методов для того, чтобы найти контекст, получить внутренний объект и найти действительный компонент.

Более того, хотя первичный ключ может получить удаленные ссылки на уникальные объектные компоненты, он не применяется с сеансовыми компонентами. Дескриптор же, напротив, может использоваться с компонентами любых типов. Благодаря этому дескриптор можно применять непротиворечиво со всеми типами компонентов. Непротиворечивость, конечно, вещь хорошая, но это еще не все. Обычно мы думаем о сеансовых компонентах как о неидентифицируемых экземплярах, поскольку они существуют лишь во время жизни клиентской сессии, но это не совсем так. Мы уже упоминали (но еще не показывали) сеансовые компоненты с состоянием, которые сохраняют информацию о состоянии в промежутках между вызовами методов. В случае применения сеансовых компонентов с состоянием два экземпляра не эквивалентны. Дескриптор позволяет нам работать с сеансовым компонентом с состоянием, деактивировать компонент, и позднее повторно активировать его.

Клиент мог бы, например, организовать при помощи сеансового компонента с состоянием обработку заказа, если этот процесс по некоторой причине прерывается. Вместо того чтобы потерять всю проделанную в сеансе работу, можно получить дескриптор из компонентного объекта и закрыть клиентское приложение. Когда пользователь будет готов продолжить заказ, на основании этого дескриптора можно будет получить ссылку на сеансовый компонент с состоянием. Обратите внимание, что в этом процессе защита от ошибок не так надежна, как в случае применения дескриптора или первичного ключа объекта с данными (entity object). Если EJB-сервер выключается или терпит крах, сеансовый компонент с состоянием теряется, и дескриптор становится бесполезен. Также существует вероятность, что сеансовый компонент по истечении времени его жизни (time out) будет удален контейнером из обслуживания и клиент больше не получит к нему доступ.

Изменения в технологии контейнеров могут сделать недействительными и дескрипторы, и первичные ключи. Если вы думаете, что технология вашего контейнера может измениться, позаботьтесь о том, чтобы учесть это ограничение. Первичные ключи получают компонентные объекты, предоставляя уникальную идентификацию экземпляров в постоянной памяти данных. Изменение в механизме постоянства, однако, может повлиять на целостность ключа.

HomeHandle

`javax.ejb.HomeHandle` по назначению похож на `javax.ejb.Handle`. Так же как и дескриптор, предназначенный для сохранения и восстановления ссылок на удаленные компоненты Object, `HomeHandle` служит для сохранения и восстановления ссылок на удаленные внутренние объекты. Другими словами, `HomeHandle` может быть сохранен и использован позднее для того, чтобы обратиться к удаленной ссылке на внутренний объект компонента точно так же, как дескриптор может быть сериализован и использован позже для обращения к удаленной ссылке на компонентный объект. В следующем коде показаны получение, сериализация и применение `HomeHandle`:

```
// Получаем каюту 100
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

// Сериализуем HomeHandle компонента Cabin
HomeHandle homeHandle = home.getHomeHandle();
FileOutputStream fos = new FileOutputStream("handle.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(homeHandle);
outStream.flush();
fos.close();
homeHandle = null;

// Десериализуем HomeHandle компонента Cabin
FileInputStream fis = new FileInputStream("handle.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
homeHandle = (HomeHandle)inStream.readObject();
fis.close();

EJBHome homeRef = homeHandle.getEJBHome();
CabinHomeRemote home2 = (CabinHomeRemote)
    PortableRemoteObject.narrow(homeRef,CabinHomeRemote.class);
```

Внутри дескриптора

Разные производители определяют свои конкретные реализации объектов `Handle` по-разному. Однако осмысление гипотетических реализаций дескрипторов позволит вам лучше понимать, как они работают. В этом примере мы определяем реализацию дескриптора для объектного компонента. Наша реализация включает поиск с помощью JNDI и использование метода `findByPrimaryKey()` таким образом, чтобы любое изменение, делающее этот ключ недействительным, не повлияло на объекты `Handle`, от него зависящие. Ниже приведен код для нашей гипотетической реализации:

```
package com.titan.cabin;

import javax.naming.InitialContext;
```

```
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.EJBObject;
import javax.ejb.Handle;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class VendorX_CabinHandle
    implements javax.ejb.Handle, java.io.Serializable {

    private Integer primary_key;
    private String home_name;
    private Properties jndi_properties;

    public VendorX_CabinHandle(Integer pk, String hn, Properties p) {
        primary_key = pk;
        home_name = hn;
        jndi_properties = p;
    }

    public EJBObject getEJBObject() throws RemoteException {
        try {
            Context ctx = new InitialContext(jndi_properties);

            Object ref = ctx.lookup(home_name);
            CabinHomeRemote home =(CabinHomeRemote)
                PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

            return home.findByPrimaryKey(primary_key);
        } catch (javax.ejb.FinderException fe) {
            throw new RemoteException("Cannot locate EJB object",fe);
        } catch (javax.naming.NamingException ne) {
            throw new RemoteException("Cannot locate EJB object",ne);
        }
    }
}
```

Дескриптор менее устойчив, чем первичный ключ, потому что он полагается на стабильность сетевой конфигурации и системы имен – IP-адреса сервера EJB и JNDI-имени компонента Home. Если изменяется сетевой адрес сервера EJB или имя, по которому идентифицируется компонент, дескриптор становится бесполезным.

Кроме этого, некоторые производители реализуют в дескрипторе механизм безопасности, который предотвращает использование его за пределами клиентского приложения, первоначально его запросившего. Как этот механизм должен работать – не ясно, но необходимо рассмотреть ограничения безопасности, которые он подразумевает, прежде чем пытаться использовать дескриптор за пределами клиента.

📖 Рабочее упражнение 5.2. EJBObject, дескриптор и первичный ключ

EJB 2.0: Локальный клиентский интерфейс

В EJB 2.0 введены понятия локальных компонентных интерфейсов, предназначенные для того, чтобы обеспечить разную семантику и разный контекст выполнения для компонентов, работающих совместно в пределах одной контейнерной системы EJB.

Два или несколько взаимодействующих компонентов обычно *совмещены* (*co-located*), другими словами, развернуты в одной контейнерной системе EJB и выполняются внутри одной виртуальной машины Java. Для связи между собой совмещенным компонентам не нужна сеть. Из-за того что они находятся в одной JVM, они могут взаимодействовать непосредственно, избегая накладных расходов Java RMI-IIOP. Однако спецификация EJB 1.1 требовала, чтобы даже совмещенные компоненты использовали для связи семантику Java RMI-IIOP. Эта спецификация не требовала применения сетевого протокола, но она требовала работы с типами Java RMI-IIOP и передачи аргументов методов по значению, а не по ссылке.

Поставщики EJB 1.1, вынужденные по каплям выжимать производительность из своих серверов, оптимизировали совмещенные компоненты. Эта оптимизация требовала, чтобы контейнер EJB участвовал в передаче вызовов одного компонента другому, но позволяла производителю обходить издержки сетевых коммуникаций. Параметры и возвращаемые значения обрабатывались контейнером внутри JVM, а не сериализовывались по сети. Однако параметры все еще должны были копироваться, а не передаваться по ссылке, что немного замедляло вызовы методов. Многие производители (если не большинство) предложили собственный переключатель, позволявший управляющему развертыванием отключить семантику копирования для совмещенных компонентов, чтобы объекты, передаваемые от одного компонента другому внутри одной контейнерной системы, могли передаваться по ссылке, а не по значению, что приводило к улучшению производительности.

Оптимизация совмещенных компонентов, в том числе и переключатели для устранения копирования параметров, в конце концов стала настолько распространенной среди производителей, что компания Sun Microsystems решила сделать эту опцию частью спецификации. Вот почему были введены локальные компонентные интерфейсы, составляющие локальный клиентский программный интерфейс (Local Client API).

Сеансовые и объектные компоненты могут по выбору реализовывать удаленный или локальный компонентные интерфейсы либо и тот и другой. Любой тип компонента (объектный, сеансовый или управляемый сообщениями) может быть совмещенным клиентом сеансового компонента или объектного компонента: например, управляемый сообщением компонент может вызывать методы совмещенных объект-

ных компонентов, используя его локальные компонентные интерфейсы. Локальный клиентский интерфейс во многом похож на удаленный клиентский интерфейс, но он менее сложен. Локальный клиентский интерфейс состоит из двух интерфейсов – локального и локального внутреннего, похожих на удаленный и удаленный внутренний интерфейсы, рассмотренные ранее в этой главе.

При объяснении локальных и локальных внутренних интерфейсов и их применения мы создадим эти интерфейсы для компонента Cabin, разработанного нами в главе 4.

Локальный интерфейс

Локальный интерфейс, как и удаленный, определяет прикладные методы компонента, которые могут быть вызваны другими совмещенными компонентами (совмещенными клиентами). Эти прикладные методы должны соответствовать сигнатурам прикладных методов, определенных в классе компонента. Например, интерфейс `CabinLocal` – это локальный интерфейс, определенный для компонента Cabin:

```
package com.titan.cabin;

import javax.ejb.EJBException;

public interface CabinLocal extends javax.ejb.EJBLocalObject {
    public String getName() throws EJBException;
    public void setName(String str) throws EJBException;
    public int getDeckLevel() throws EJBException;
    public void setDeckLevel(int level) throws EJBException;
    public int getShipId() throws EJBException;
    public void setShipId(int sp) throws EJBException;
    public int getBedCount() throws EJBException;
    public void setBedCount(int bc) throws EJBException;
}
```

Интерфейс `CabinLocal` в основном такой же, как и интерфейс `CabinRemote`, разработанный нами в главе 4, но с парой ключевых отличий. Важнее всего то, что интерфейс `CabinLocal` расширяет интерфейс `javax.ejb.EJBLocalObject`, а его методы не генерируют исключение `java.rmi.RemoteException`.

Локальные интерфейсы должны расширять интерфейс `javax.ejb.EJBLocalObject`, тогда как удаленные интерфейсы должны расширять интерфейс `javax.ejb.EJBObject`:

```
package javax.ejb;

import javax.ejb.EJBException;
import javax.ejb.RemoteException;

public interface EJBLocalObject {
```

```
    public EJBLocalHome getEJBLocalHome() throws EJBException;
    public Object getPrimaryKey() throws EJBException;
    public boolean isIdentical(EJBLocalObject obj) throws EJBException;
    public void remove() throws RemoveException, throws EJBException;
}
```

В интерфейсе `EJBLocalObject` определено несколько методов, которые должны быть вам знакомы из предыдущих разделов. Метод `getEJBLocalHome()` возвращает локальный внутренний объект, метод `getPrimaryKey()` возвращает первичный ключ (только для объектных компонентов); метод `isIdentical()` сравнивает два локальных компонентных объекта, а метод `remove()` удаляет компонент. Эти методы работают точно так же, как соответствующие им методы интерфейса `javax.ejb.EJBObject`. Кроме того, важно указать, что `EJBLocalObject`, в отличие от `EJBObject`, не расширяет интерфейс `java.rmi.Remote`, т. к. он не является удаленным объектом.

Вы, возможно, заметили, что `EJBLocalObject`, в отличие от `EJBObject`, не определяет метод `getHandle()`. Локальные интерфейсы не определяют метод `getHandle()`, потому что `javax.ejb.Handle` не нужен, если и клиент, и компоненты расположены в одной контейнерной системе EJB. Дескриптор – это сериализуемая ссылка, упрощающая получение ссылки для удаленного клиента на компонент, расположенный на удаленном сетевом узле. Поскольку совмещенные компоненты расположены в одной контейнерной системе, а не распределены по сети, объект `Handle` не требуется.

`EJBLocalObject` и локальные интерфейсы, расширяющие его, не генерируют исключение `java.rmi.RemoteException`. Эти интерфейсы применяются для совмещенных компонентов, расположенных в одной JVM, а не для доступа к удаленным компонентам, поэтому исключение `RemoteException`, используемое для индикации о сетевых сбоях и частичных отказах удаленной системы, не требуется. Вместо этого локальные интерфейсы и `EJBLocalObject` генерируют исключение `EJBException`. Последнее автоматически возбуждается контейнером при вызове локальных методов интерфейса, если происходит какая-либо системная ошибка контейнера или ошибка транзакции, приводящая к уничтожению экземпляра компонента.

`EJBException` представляет собой подтип `java.lang.RuntimeException`, и, следовательно, это неперехватываемое (unchecked) исключение. Неперехватываемые исключения не надо объявлять в секции `throw` локальных компонентных интерфейсов, не требуется также, чтобы клиент явно обрабатывал их с помощью блоков `try/catch`. Однако мы решили объявить `EJBException` в сигнатурах методов интерфейса `CabinLocal` для того, чтобы сообщить клиентскому приложению, что этот тип исключения возможен.

Локальный внутренний интерфейс

Локальный внутренний интерфейс, как и удаленный внутренний интерфейс, определяет методы жизненного цикла компонента, которые могут быть вызваны другими совмещенными компонентами (совмещенными клиентами). Методы жизненного цикла локального внутреннего интерфейса содержат методы для поиска, создания и удаления, похожие на методы удаленного внутреннего интерфейса. `CabinHomeLocal`, локальный внутренний интерфейс компонента Cabin, показан в следующем листинге:

```
package com.titan.cabin;

import javax.ejb.EJBException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHomeLocal extends javax.ejb.EJBLocalHome {

    public CabinLocal create(Integer id)
        throws CreateException, EJBException;

    public CabinLocal findByPrimaryKey(Integer pk)
        throws FinderException, EJBException;
}
```

Интерфейс `CabinHomeLocal` очень похож на свой аналог `CabinHomeRemote`, разработанный нами в главе 4. Однако `CabinHomeLocal` расширяет `javax.ejb.EJBLocalHome` и не генерирует `RemoteException` в его конструирующих и поисковых методах. Вы, возможно, также заметили, что тип, возвращаемый из методов `create()` и `findByPrimaryKey()`, – это интерфейс CabinLocal, а не удаленный интерфейс компонента Cabin. Конструирующие и поисковые методы локальных внутренних интерфейсов всегда возвращают компонентные объекты, реализующие локальный интерфейс этого компонента.

Локальные интерфейсы всегда должны расширять интерфейс `EJBLocalHome`, который намного проще своего удаленного аналога `EJBHome`:

```
package javax.ejb;

import javax.ejb.RemoveException;
import javax.ejb.EJBException;

public interface EJBLocalHome {
    public void remove(Object primaryKey)
        throws RemoveException, EJBException;
}
```

В отличие от `EJBHome`, `EJBLocalHome` не предоставляет методов доступа к `EJBMetaData` и `HomeHandle`. Объект `EJBMetaData`, используемый, прежде всего, визуальными инструментальными средствами разработки, не нужен для совмещенных компонентов. Точно так же `HomeHandle` не имеет отношения к совмещенным клиентским компонентам даже в большей

степени, чем дескриптор, из-за того что совмещенным компонентам не требуются специальные сетевые ссылки. В EJBLocalHome определен метод remove(), принимающий в качестве параметра первичный ключ. Этот метод работает точно так же, как соответствующий ему метод в удаленном интерфейсе EJBObject.

Дескриптор развертывания

Когда компонент использует локальные компонентные интерфейсы, они должны быть объявлены в XML-дескрипторе развертывания. Это достаточно простое дело. Изменения, связанные с ним, выделены далее:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.cabin.CabinHomeRemote</home>
            <remote>com.titan.cabin.CabinRemote</remote>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
            <ejb-class>com.titan.cabin.CabinBean</ejb-class>
```

Кроме добавления элементов <local-home> и <local>, элемент <ejb-ref> был заменен на элемент <ejb-local-ref>, указывая, что вместо удаленного используется локальный компонентный объект:

```
<ejb-local-ref>
    <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>com.titan.cabin.CabinHomeLocal</local-home>
    <local>com.titan.cabin.CabinLocal</local>
</ejb-local-ref>
```

Применение локального клиентского интерфейса

Мы можем легко перепроектировать компонент TravelAgent, разработанный в главе 4, так чтобы он вместо удаленных интерфейсов компонента Cabin использовал его локальные интерфейсы:

```
public String [] listCabins(int shipID, int bedCount) {

    try {
        javax.naming.Context jndiContext = new InitialContext();
        CabinHomeLocal home = (CabinHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");
        Vector vect = new Vector();
        for (int i = 1; ; i++) {
            Integer pk = new Integer(i);
            CabinLocal cabin;
```

```
                try {
                    cabin = home.findByPrimaryKey(pk);
                } catch(javax.ejb.FinderException fe) {
                    break;
                }
                // проверяем, совпадают ли число коек и идентификаторы судов
                if (cabin.getShipId() == shipID &&
                    cabin.getBedCount() == bedCount) {
                    String details =
                    i+",'"+cabin.getName()+"',"+cabin.getDeckLevel();
                    vect.addElement(details);
                }
            }

            String [] list = new String[vect.size()];
            vect.copyInto(list);
            return list;

        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }
```

Полужирный текст показывает три необходимые изменения, сделанные в коде. Наиболее важное изменение – это применение в компоненте Cabin локальных компонентных интерфейсов вместо удаленных. Кроме этого, получая внутренний объект компонента Cabin от JNDI ENC, мы не обязаны использовать метод `PortableRemoteObject.narrow()`. Это происходит потому, что мы не обращаемся к внутреннему компоненту через сеть. Мы обращаемся к нему изнутри той же самой JVM и поэтому мы знаем, что клиентом является клиент Java и что не применяется никакой сетевой протокол. Устранение этого метода привело к тому, что данный код стал выглядеть намного проще. Кроме того, мы изменили блок `try/catch` так, чтобы он вместо исключения `EJBException` перехватывал `javax.naming.NamingException`, генерируемое любым из методов локального компонентного интерфейса. В этом случае проще разрешить указанным исключениям передаваться непосредственно в контейнер, где они могут быть обработаны лучше. Обработка особых ситуаций подробно рассмотрена в главе 14.

📖 Рабочее упражнение 5.3. Локальные компонентные интерфейсы

Когда применять локальные компонентные интерфейсы

Объектные и сеансовые компоненты могут использовать либо локальные, либо удаленные компонентные интерфейсы, либо оба типа, – для того чтобы компонент был доступен как удаленным, так и локальным клиентам. Каждый раз, намереваясь обеспечить взаимодействие компонентов друг с другом в пределах одной контейнерной системы,

необходимо серьезно рассмотреть применение локальных компонентных интерфейсов, поскольку они, вероятно, будут эффективнее, чем удаленные компонентные интерфейсы. При использовании локальных клиентских интерфейсов не применяется никакая сетевая инфраструктура и не происходит копирование параметров.

Поскольку локальные клиенты находятся в той же самой JVM, в сетевых соединениях нет никакой надобности. Когда совмещенный клиентский компонент вызывает метод локального компонентного объекта, контейнер служит для проведения транзакций и управления безопасностью, но затем перенаправляет вызов непосредственно целевому экземпляру компонента. Без каких-либо сетевых соединений. С очень небольшими накладными расходами. Полное отсутствие сетевых соединений может до предела увеличивать быстродействие цикла RMI, в связи с чем это очень желательно. Однако в случае применения локального клиентского интерфейса пропадает прозрачность расположения компонентных ссылок. Другими словами, нельзя переместить компонент на другой сетевой сервер из-за того, что он должен остаться совмещенным. А удаленный клиентский интерфейс позволяет перемещать компоненты с одного сервера на другой без изменения кода компонента. Другими словами, удаленный клиентский интерфейс обеспечивает большую прозрачность расположения: нам не надо знать, где расположен компонент, для того чтобы вызывать его методы.

Кроме того, локальный клиентский интерфейс передает объекты от одного компонента другому по ссылке (рис. 5.5). Это означает, что на

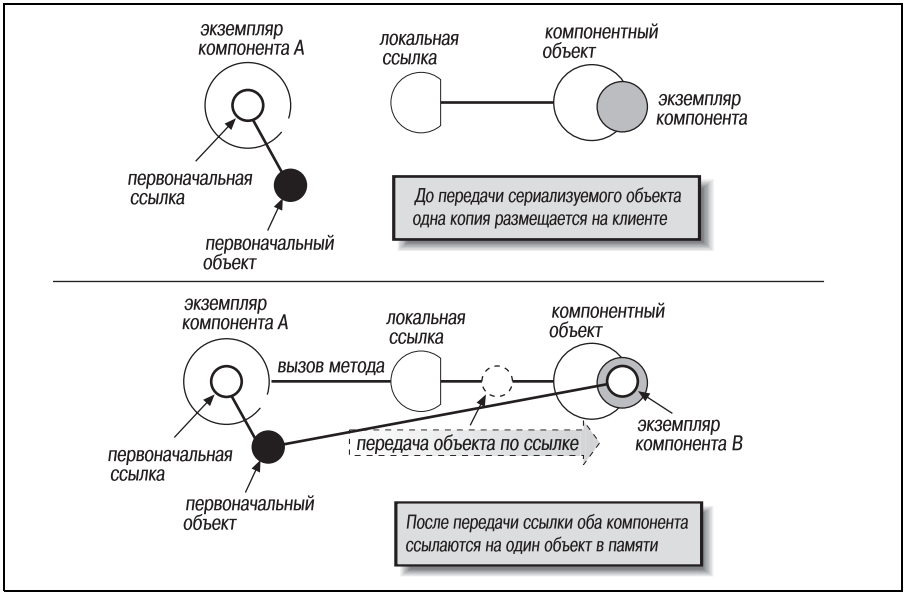

Рис. 5.5. Передача по ссылке с использованием локального клиентского интерфейса

объект, переданный от компонента A компоненту B, будут ссылаться оба эти компонента, поэтому, если B изменит свое значение, эти изменения будут видны и компоненту A.

В случае применения удаленного клиентского интерфейса объекты (параметры или возвращаемые значения) всегда копируются, поэтому изменения одной копии не повлияют на другую (см. рис. 5.1).

Передача по ссылке чревата некоторыми довольно опасными ситуациями в тех случаях, если компоненты, совместно использующие объектную ссылку, не были разработаны достаточно тщательно. В большинстве случаев лучше передать неизменные объекты без предварительного копирования. В главе 16 рассмотрено применение неизменных зависимых объектов, также хорошо подходящих для передачи по ссылке. Однако если вы собираетесь передать объект, не являющийся неизменяемым, объект должен быть предварительно скопирован. Это правило применимо во всех случаях, когда передаваемые объекты не изменяются никакими компонентами, кроме тех, от которых они пришли.

Действительно ли необходимы локальные компонентные интерфейсы?

Некоторые производители утверждают, что локальные компонентные интерфейсы никогда не нужно ставить на первое место и что они лишь вносят многочисленные сложности, не добавляя платформе EJB никаких реальных функциональных возможностей. Это – разумный аргумент, принимая во внимание, что большинство производителей EJB 1.1 уже оптимизировали удаленный клиентский интерфейс для совмещенных компонентов. Как мы видели раньше, эти производители не используют сетевые соединения для взаимодействий между совмещенными компонентами и предоставили переключатели копирования параметров.

Так зачем же локальный клиентский интерфейс вообще нужен? Одной из альтернатив было исправление спецификации удаленного клиентского интерфейса с учетом оптимизации контейнерного совмещения, при том, что эта оптимизация становилась стандартным конфигурационным атрибутом дескриптора развертывания. Единственная проблема при таком решении заключается в семантике. Удаленные интерфейсы расширяют `java.rmi.Remote`, который предназначен специально для удаленных объектов. Кроме того, всем подтипам интерфейса `java.rmi.Remote` необходимо возбуждать в своих методах исключения с типами `java.rmi.RemoteException`. Разработчикам может оказаться нелегко уловить разницу между совмещенным и удаленным компонентными объектами, важную в случаях, когда один из них передает объекты по ссылке, а другой – по значению.

Однако разработчикам EJB может также оказаться нелегко привыкнуть к локальным компонентным интерфейсам и использовать их эффективно. Применение локальных компонентных интерфейсов ограничивает нас единственной JVM и не позволяет перемещать компоненты с одного контейнера на другой по своему желанию. Каждый из аргументов «за» и «против» локальных компонентных интерфейсов имеет свои достоинства. Локальные клиентские интерфейсы существуют независимо от нашего желания, и необходимо научиться использовать их правильно.

6

EJB 2.0 CMP: Основы постоянства

Обзор

В главе 4 мы начали разрабатывать несколько простых компонентов, пропуская многие детали, относящиеся к их созданию. Здесь мы дадим полный обзор процесса разработки объектных компонентов. Объектные компоненты моделируют прикладные понятия, которые могут быть выражены существительными. Это скорее эмпирическое правило, а не требование, но оно помогает определить, когда прикладное понятие следует реализовывать в виде объектного компонента. Со средней школы мы знаем, что существительные – это слова, которые описывают человека, место или предмет. Понятия «человек» и «место» достаточно очевидны: компонент «Человек» может представлять клиента или пассажира, а компонент «Место» может представлять город или «место назначения». Точно так же объектные компоненты часто представляют «предметы»: реальные объекты, подобные судам, кредитным карточкам и т. д. Объектный компонент может даже представлять достаточно абстрактную вещь, вроде резервирования билетов. Объектные компоненты описывают и состояние, и поведение реальных объектов и позволяют разработчикам включать в них данные и прикладные правила, связанные с особыми понятиями. К примеру, компонент «Пассажир» содержит данные и прикладные правила, связанные с пассажиром. Это делает возможным логичное и безопасное связывание данных с понятием, которым необходимо манипулировать.

В примере с системой «Титан» можно выделить сотни прикладных понятий, которые представляют собой существительные и поэтому, ес-

тественно, могут быть смоделированы объектными компонентами. Мы уже видели простой компонент «Каюта» в главе 4, а здесь мы создадим компоненты «Пассажир» и «Адрес». Очевидно, что в примере могли бы использоваться компоненты «Круиз», «Резервирование» и многие другие. Каждое из этих прикладных понятий представляет данные, которые необходимо отслеживать и которыми возможно управлять.

Элементы представляют данные в базе данных, поэтому изменения объектного компонента приводят к изменениям в базе данных. Имеется много причин для применения объектных компонентов вместо непосредственного доступа к базе данных. Объектные компоненты дают программистам более простой механизм для доступа и изменения данных. Значительно проще, например, поменять имя клиента, вызвав метод `Customer.setName()`, чем выполнять для базы данных команду SQL. Кроме того, хранение данных в объектных компонентах упрощает повторное использование программы. После того как объектный компонент определен, он непротиворечивым образом может применяться во всей системе «Титан». Понятие «клиент», например, используется во многих модулях системы «Титан», включая заказ, планирование и маркетинг. Компонент «Пассажир» предоставляет «Титану» полный доступ к информации о пассажире, гарантируя таким образом логичность и простоту доступа к информации. Представление данных в виде объектных компонентов может сделать разработку легче и рентабельнее.

При создании нового компонента в базу данных должна быть вставлена новая запись и создан экземпляр компонента, связанный с этими данными. По мере использования компонента его состояние меняется, и изменения должны быть синхронизированы с данными в базе данных: записи должны вставляться, модифицироваться и удаляться. Этот процесс координации данных, представляющих экземпляр, с базой данных называется *постоянством* (*persistence*).

Существуют два основных типа объектных компонентов, различающихся реализацией механизма постоянства: компоненты с постоянством, управляемым контейнером, и компоненты, поддерживающие постоянство самостоятельно. В случае с *постоянством, управляемым контейнером* (*container-managed persistence*), контейнер знает, как постоянство экземпляра и поля отношений проецируются на базу данных, и автоматически вставляет, модифицирует и удаляет данные, связанные с объектами, из базы данных. В случае *объектных компонентов, самостоятельно реализующих постоянство* (*bean-managed persistence*), вся эта работа выполняется вручную разработчиком компонента, который должен написать код для манипуляции базой данных. Контейнер EJB сообщает экземпляру компонента, когда безопасно вставлять, модифицировать и удалять его данные из базы данных, но не предоставляет никакой другой помощи. Экземпляр компонента

делает всю работу по обеспечению своего постоянства самостоятельно. Постоянство, управляемое компонентом, рассматривается в главе 10.

В EJB 2.0 управляемое контейнером постоянство подверглось настолько сильным изменениям, что оно больше не обеспечивает обратной совместимости с EJB 1.1. По этой причине поставщики EJB 2.0 должны поддерживать модели постоянства, управляемого контейнером, определенные в обеих спецификациях. Модель EJB 1.1 все еще поддерживается, для того чтобы прикладные разработчики могли перенести свои существующие приложения на новую платформу EJB 2.0 как можно менее болезненно. Предполагается, что все новые объектные компоненты и новые приложения будут использовать постоянство, управляемое контейнером, реализованное в версии EJB 2.0, а не EJB 1.1. И хотя постоянство, управляемое контейнером, в EJB 1.1 рассматривается в этой книге, его следует избегать, если только вы не обременены необходимостью поддержания существующих систем EJB 1.1. Постоянство, управляемое контейнером, реализованное в EJB 1.1, рассматривается в главе 9.

В этой и двух последующих главах рассматривается создание объектных компонентов, которые используют управляемое контейнером постоянство EJB 2.0. В EJB 2.0 данные, связанные с компонентом, могут быть намного более сложными, чем это было возможно в EJB 1.1. В EJB 2.0 компоненты с постоянством, управляемым контейнером, могут иметь связи с другими компонентами, которые недостаточно хорошо поддерживались в старой версии – в результате поставщики иногда предлагали частные решения, которые не были переносимы. Кроме того, в EJB 2.0 компоненты с постоянством, управляемым контейнером, могут быть более мелкими – для того чтобы было легче моделировать понятия вроде адреса, рейса или салона.

В данной главе разрабатываются два очень простых объектных компонента – Client и Address, – на примере которых будет объясняться, как в Enterprise JavaBeans 2.0 компоненты с постоянством, управляемым контейнером, определяются и обрабатываются во время выполнения. Компонент Client имеет связи с несколькими другими объектами, такими как Address, Phone, CreditCard, Cruise, Ship, Cabin и Reservation. В следующих нескольких главах мы посмотрим, как использовать мощную поддержку связей между компонентами в EJB 2.0 для и постараемся понять их ограничения. Кроме того, в главе 8 мы изучим язык запросов Enterprise JavaBeans (EJB QL), применяемый для описания «поведения» методов поиска и новых методов выборки во время выполнения.

Для постоянства, управляемого контейнером, в EJB 2.0 общепринятым является обозначение *CMP 2.0*, и *CMP 1.1* – для постоянства, управляемого контейнером в EJB 1.1.

Абстрактная программная модель

В CMP 2.0 контейнер объектных компонентов автоматически управляет их состоянием. Контейнер заботится о регистрации компонентов в транзакциях и поддерживает их состояние в базе данных. Разработчик компонентов описывает атрибуты и связи компонента с помощью *виртуальных* (*virtual*) полей постоянства и отношений. Они названы виртуальными полями, потому что разработчик не определяет их явно. Вместо этого в классе компонента объявляются абстрактные методы доступа (get и set). Реализация этих методов создается во время развертывания приложения при помощи инструментальных средств, предоставляемых поставщиком EJB. Важно запомнить, что термины *поле отношения* (*relationship field*) и *поле постоянства* (*persistence field*) относятся к абстрактным методам доступа, а не к настоящим полям, объявленным в классе. Это терминологическое соглашение, принятое в EJB, должно показаться вам удобным.

Компонент Customer, показанный на рис. 6.1, содержит шесть методов доступа. Первые четыре применяются для чтения и изменения имени и фамилии клиента. Они представляют собой примеры полей постоянства: простых непосредственных атрибутов компонента. Последние два метода доступа получают и устанавливают ссылки на компонент Address через его локальный интерфейс `AddressLocal`. Это пример поля отношений с именем `homeAddress`.

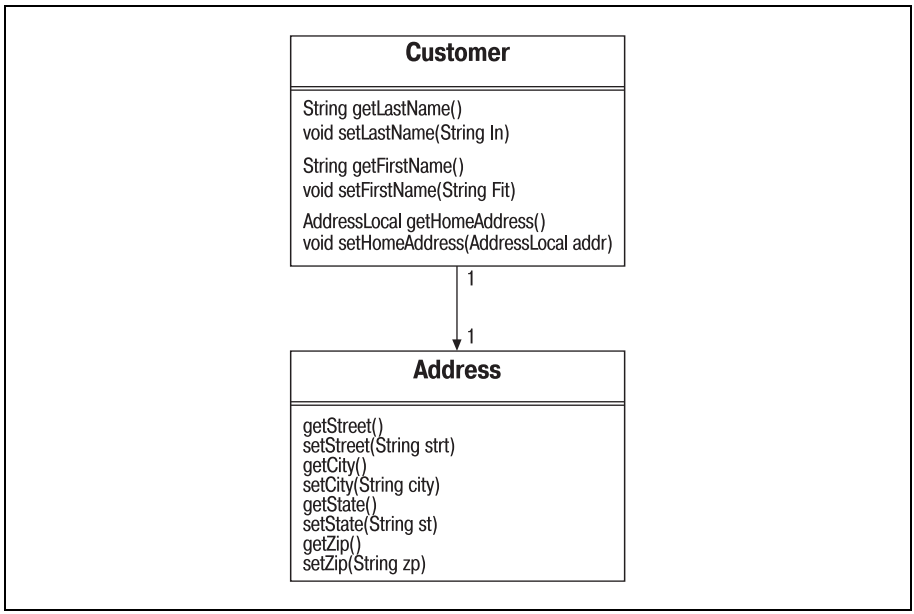

Рис. 6.1. Диаграмма классов компонентов Customer и Address

Абстрактная схема постоянства

Объектные компоненты в CMP 2.0 определяются с помощью абстрактных методов доступа, которые представляют виртуальные поля постоянства и отношений. Как уже упоминалось, сами настоящие поля не объявляются в классах компонентов. Но характеристики этих полей детально описываются в XML-дескрипторе развертывания, используемом этим компонентом. Абстрактная схема постоянства представляет собой набор XML-элементов в дескрипторе развертывания, описывающих поля отношений и поля постоянства. Вместе с абстрактной программной моделью (абстрактными методами доступа) и частично с помощью администратора развертывания инструментальные средства контейнера располагают достаточной информацией для установления связей данного компонента с другими компонентами в базе данных.

Инструментальные средства контейнера и постоянство

Инструментальные средства развертывания контейнера в числе прочего отвечают за создание конкретной реализации абстрактных объектных компонентов. Реальные классы, генерируемые инструментами контейнера, называются *классами постоянства* (*persistence classes*). Экземпляры этих классов отвечают за работу с контейнером по чтению и записи данных между компонентом и базой данных во время выполнения. После создания классов с постоянством последние могут быть развернуты в контейнере EJB. Контейнер сообщает *постоянным экземплярам* (*persistence instances*), или экземплярам класса с постоянством, момент, когда можно считывать и записывать данные в базу данных. Постоянные экземпляры выполняют процедуры записи и чтения самым оптимальным для этой базы данных образом.

К классам с постоянством будет добавлена логика доступа к конкретной базе данных. Например, в состав продукта EJB может входить контейнер, способный связывать объектный компонент с определенной базой данных, например с реляционной базой данных Oracle или объектной базой данных POET. Эта особенность позволяет классам с постоянством использовать «родную» оптимизацию, специфичную для данного типа (или торговой марки) базы данных, схемы и конфигурации. Для дальнейшего улучшения производительности классы с постоянством могут также использовать такую оптимизацию, как отложенная загрузка или оптимистическая блокировка.

Инструменты контейнера полностью реализуют процедуры доступа к базе данных во время развертывания и встраивают их в классы с постоянством. Это означает, что разработчики компонентов не должны сами реализовывать логику доступа к базам данных, что сберегает им много времени. Это также приводит к улучшенной производительности объектных компонентов благодаря тому, что реализация оптимизи-

рована. В качестве создателя объектных компонентов при работе с компонентами CMP 2.0 вам никогда не придется иметь дело с кодом для доступа к базам данных. Фактически, возможно, вам не придется иметь дело с классами с постоянством, содержащими такую логику, потому что они автоматически генерируются средствами контейнера. В большинстве случаев их исходный код недоступен разработчику компонентов.

На рис. 6.2 и 6.3 показаны инструментальные средства двух контейнеров, которые оба используются для связывания объектного компонента Customer и реляционной базы данных.

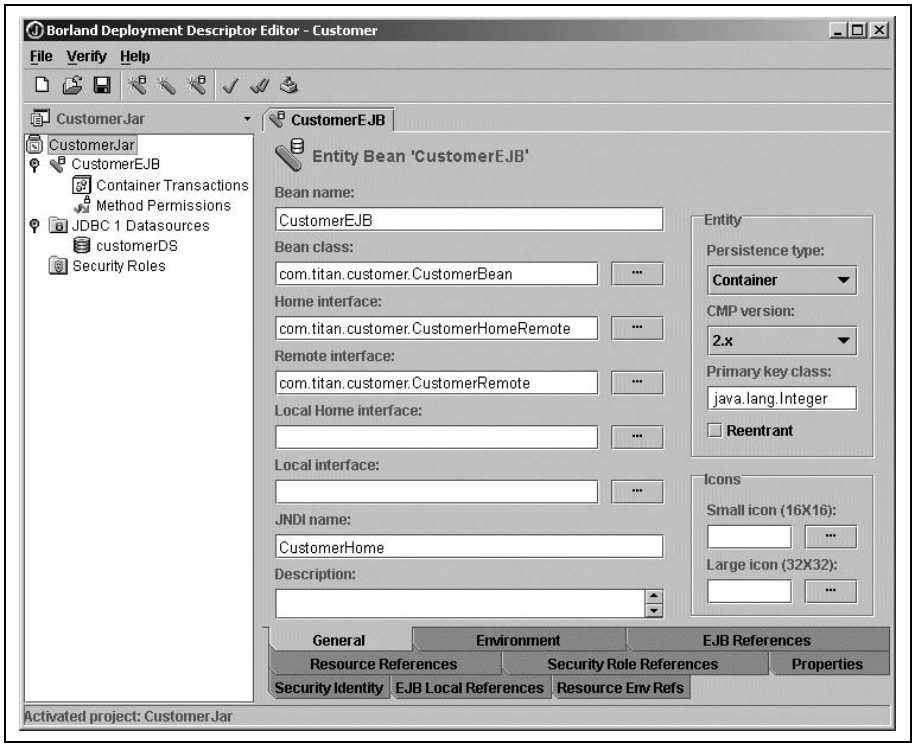

Рис. 6.2. Средство развертывания AppServer фирмы Borland

Компонент Customer

В следующем примере мы создадим простой объектный компонент CMP 2.0 – Customer. Здесь он моделирует понятие «пассажир круиза», но и его разработка, и использование применимы ко многим областям коммерции.

Для того чтобы проиллюстрировать принципы, обсуждаемые в каждом разделе, на протяжении всей этой главы компонент Customer

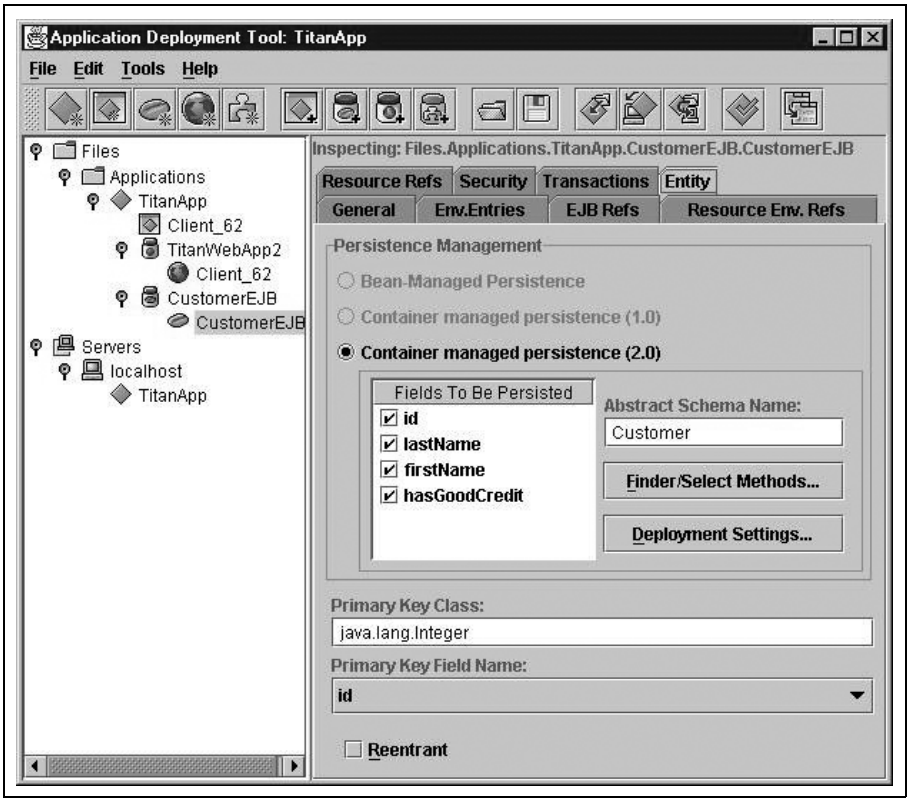

Рис. 6.3. Средство развертывания J2EE 1.3 SDK

будет расширяться, а его сложность – возрастать. В данном разделе мы лишь познакомимся с объектным компонентом и несколькими основными принципами, относящимися к его разработке, упаковке и развертыванию. С целью упростить понимание предпримем краткий обзор принципов, которые подробно обсуждаются далее в этой главе.

Таблица Customer

Хотя CMP 2.0 и не зависит от конкретной базы данных, примеры, приводимые в книге, предполагают, что мы работаем с реляционной базой данных. Это означает, что нам понадобится таблица `CUSTOMER` – для того чтобы получить сведения о нашем клиенте. Таблица реляционной базы данных определяется на языке SQL следующим образом:

```
CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY NOT NULL,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20)
)
```

Класс CustomerBean

Класс `CustomerBean` представляет собою абстрактный класс, при помощи которого инструменты контейнера будут создавать конкретную реализацию класса с постоянством, выполняемого внутри контейнера EJB. Механизмы, посредством которых инструменты контейнеров создают класс с постоянством, варьируются, но большинство поставщиков генерируют производный класс от абстрактного класса, предоставленного разработчиком компонента (рис. 6.4).

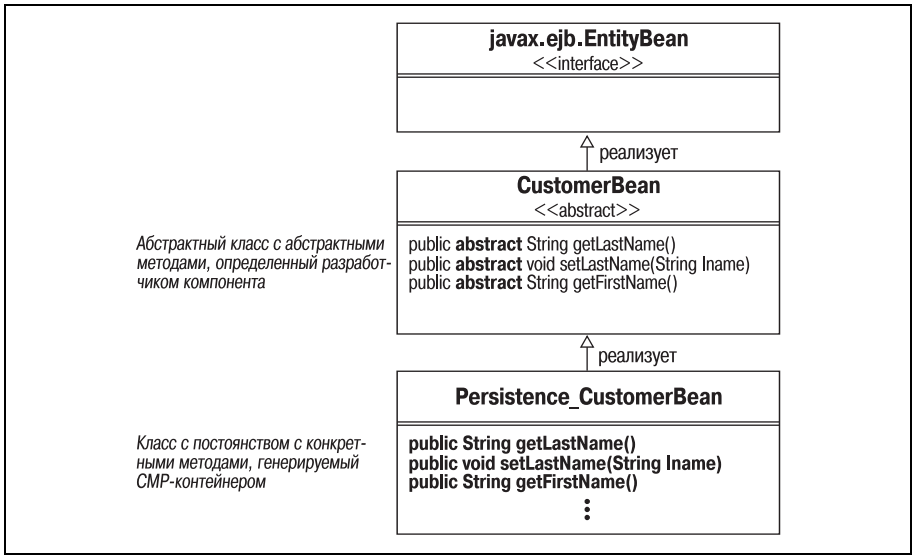

Рис. 6.4. Инструмент контейнера обычно расширяет класс компонента

Класс компонента должен объявить методы доступа (set и get) для каждого из полей постоянства и отношений, заданных в абстрактной схеме постоянства в дескрипторе развертывания. Для полного описания схемы постоянства компонента инструменту контейнера будут нужны и абстрактные методы доступа (определенные в классе компонента), и XML-элементы дескриптора развертывания. В этой книге класс объектного компонента всегда определяется до элементов XML, т. к. это наиболее естественный подход при разработке объектных компонентов.

Здесь приведено очень простое определение класса `CustomerBean`, создаваемого и упаковываемого для развертывания разработчиком компонента.

```
package com.titan.customer;

import javax.ejb.EntityContext;

public abstract class CustomerBean implements javax.ejb.EntityBean {
```

```
    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
    }

    // Абстрактные методы доступа

    public abstract Integer getId();
    public abstract void setId(Integer id);

    public abstract String getLastName();
    public abstract void setLastName(String lname);

    public abstract String getFirstName();
    public abstract void setFirstName(String fname);

    // Стандартные методы обратного вызова

    public void setEntityContext(EntityContext ec){}
    public void unsetEntityContext(){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
}
```

Класс `CustomerBean` определен как абстрактный класс. Этого требует модель CMP 2.0 с целью подчеркнуть тот факт, что компонент `CustomerBean` не развертывается непосредственно в системе контейнера. Поскольку абстрактные классы не могут порождать свои экземпляры, для того чтобы быть развернутым, класс компонента должен быть использован для создания производного класса с постоянством, генерируемого средством развертывания. Кроме того, сами методы доступа объявлены абстрактными, что требуется для реализации их инструментом контейнера.

Класс `CustomerBean` расширяет интерфейс `javax.ejb.EntityBean`, определяющий несколько методов обратного вызова. В их числе: `setEntityContext()`, `unsetEntityContext()`, `ejbLoad()`, `ejbStore()`, `ejbActivate()`, `ejbPassivate()` и `ejbRemove()`. Эти методы важны для оповещения экземпляра компонента о событиях, происходящих в его жизненном цикле, но нам пока нет необходимости думать о них. Подробно эти методы обсуждаются в главе 11.

Первым в классе компонентов является метод `ejbCreate()`, принимающий ссылку на объект `Integer` в качестве своего единственного аргумента. Метод `ejbCreate()` вызывается в случае, когда удаленный клиент вызывает метод внутреннего интерфейса компонента. Этот прин-

цип должен быть вам знаком, поскольку таким же образом метод ejbCreate() функционировал и в компоненте Cabin, разработанном в главе 4. Метод ejbCreate() отвечает за инициализацию всех полей постоянства перед созданием компонента. В этом первом примере метод ejbCreae() применяется для задания начального значения полю постоянства id, представленного методами доступа setId() и getId().

Почему метод ejbCreate() возвращает null

В EJB 1.0 – первой версии EJB – метод ejbCreate() в постоянстве, реализуемом контейнером, был объявлен как возвращающий тип void, тогда как метод ejbCreate() в постоянстве, поддерживаемом компонентом, возвращал тип первичного ключа с действительным значением null.

В EJB 1.1 тип значения, возвращаемого методом ejbCreate(), был изменен с void на тип первичного ключа с целью обеспечения возможности наследования, т. е. для того чтобы упростить компонентам, самостоятельно реализующим постоянство, расширение компонентов, управляемых контейнером. В EJB 1.0 это было невозможно из-за того, что язык Java не позволяет переопределять методы с различными типами возвращаемых значений. Изменив это определение так, чтобы компоненты, самостоятельно реализующие постоянство, могли расширять компоненты, управляемые контейнером, EJB 1.1 дала возможность производителям поддерживать постоянство, управляемое контейнером, путем расширения компонентов, управляемых контейнером, созданными компонентами, поддерживающими постоянство, – достаточно простое решение сложной проблемы.

С появлением CMP 2.0 этот несложный трюк стал для производителей EJB менее полезным, чем раньше. Абстрактная схема постоянства компонентов EJB CMP 2.0 во многих случаях слишком сложна для простых контейнеров BMP. Однако она осталась в составе программной модели для обеспечения обратной совместимости и возможности наследования постоянства, реализуемого компонентом, если возникнет такая необходимость.

Метод возвращает результат типа Integer, являющийся первичным ключом этого компонента. Первичный ключ – это уникальный идентификатор, способный принимать различные формы, например, быть «оберткой» примитивных типов и пользовательских классов. В этом случае первичный ключ (целого типа) связывается с полем ID таблицы Customer. Это станет понятным, когда мы определим XML-дескриптор развертывания. Хотя тип результата метода ejbCreate() представляет собой первичный ключ, на самом деле метод ejbCreate() возвращает null. Контейнер EJB и класс с постоянством извлекают первичный ключ из компонента по мере необходимости. За разъяснением типа ре-

зультата метода ejbCreate() обратитесь к врезке «Почему метод ejbCreate() возвращает null».

Метод ejbPostCreate() предназначен для выполнения инициализации после создания объектного компонента, но прежде чем он начнет обрабатывать запросы клиента. Этот метод обычно применяется для выполнения действий над полями отношений компонента, которые могут быть выполнены только после вызова метода ejbCreate() и добавления компонента в базу данных. Для каждого метода ejbCreate() должен существовать соответствующий ему метод ejbPostCreate(), имеющий те же имена методов и аргументов, но возвращающий значение void. Такое сочетание ejbCreate() и ejbPostCreate() гарантирует, что контейнер корректно вызывает эти методы совместно. Более подробно мы объясним использование метода ejbPostCreate() позднее. Сейчас это не требуется, поэтому оставим его реализацию пустой.

Абстрактные методы доступа (setLastName(), getLastName(), setFirstName(), getFirstName()) класса CustomerBean представляют поля постоянства. Эти методы определены как пустые абстрактные методы. Как уже упоминалось, когда компонент будет обрабатываться средствами контейнера, эти методы будут реализованы классом с постоянством на основе абстрактной схемы постоянства (XML-элементы дескриптора развертывания), конкретного контейнера EJB и используемой базы данных. В основном эти методы извлекают и обновляют значения в базе данных и не реализуются разработчиком компонента.

Удаленный интерфейс

Поскольку к компоненту Customer будут обращаться клиенты, находящиеся за пределами контейнера, ему необходим интерфейс CustomerRemote. Удаленный интерфейс определяет прикладные методы, моделирующие внешний вид имитируемого прикладного понятия, т. е. его поведение и данные, которые должны быть предоставлены клиентскому приложению. Здесь приведен удаленный интерфейс CustomerRemote:

```
package com.titan.customer;
import java.rmi.RemoteException;

public interface CustomerRemote extends javax.ejb.EJBObject {

    public String getLastName() throws RemoteException;
    public void setLastName(String lname) throws RemoteException;

    public String getFirstName() throws RemoteException;
    public void setFirstName(String fname) throws RemoteException;
}
```

Все методы, определяемые в удаленном интерфейсе, должны совпадать с определениями методов в классе компонента. В нашем случае

методы доступа интерфейса CustomerRemote соответствуют методам доступа к постоянным полям класса CustomerBean. Если методы удаленного интерфейса соответствуют методам полей с постоянством, клиент имеет непосредственный доступ к полям постоянства в объектном компоненте.

Мы не должны обеспечивать соответствие между методами абстрактного доступа в классе компонента с методами удаленного интерфейса. Фактически рекомендуется, чтобы удаленный интерфейс был максимально независимым от абстрактной программной модели.

В то время как удаленные методы могут соответствовать постоянным полям класса компонента, спецификация запрещает, чтобы удаленные методы соответствовали полям отношений, связанным с другими объектными компонентами. Кроме того, удаленные методы не могут изменять никакие поля постоянства, управляемого контейнером, являющиеся частью первичного ключа компонента. Заметьте, что удаленный интерфейс не определяет метод setId(), который позволил бы изменить первичный ключ.

Удаленный внутренний интерфейс

Удаленный внутренний интерфейс любого объектного компонента используется для создания, поиска и удаления компонентов из контейнера EJB. Каждый отдельный тип объектного компонента может иметь свои собственные удаленный внутренний интерфейс, локальный внутренний интерфейс или оба сразу. Как мы видели в главе 5, удаленный и локальный внутренние интерфейсы по существу выполняют одну и ту же функцию. Внутренние интерфейсы определяют методы трех основных типов: внутренние прикладные методы, необязательные конструирующие методы и один или несколько поисковых методов.[1] Методы create() действуют в качестве удаленных конструкторов и определяют способ создания компонентов. Наш удаленный внутренний интерфейс предоставляет только один метод create(), соответствующий методу ejbCreate() в классе компонента. Поисковый метод применяется для нахождения определенного компонента Customer с помощью первичного ключа в качестве уникального идентификатора. Следующая программа содержит полное определение интерфейса CustomerHomeRemote:

```
package com.titan.customer;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
```

[1] В главе 15 рассказано, когда не следует объявлять конструирующие методы во внутреннем интерфейсе.

```
public interface CustomerHomeRemote extends javax.ejb.EJBHome {

    public CustomerRemote create(Integer id)
        throws CreateException, RemoteException;

    public CustomerRemote findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;

}
```

К имени метода `create()` может быть добавлено любое окончание для его уточнения при переопределении аргументов метода. Это полезно при наличии двух разных методов `create()`, принимающих аргументы одинаковых типов. Например, мы могли бы объявить два метода `create()` для компонента Customer, оба объявляющих аргументы типов `Integer` и `String`. Аргумент типа `String` мог бы быть в первом случае номером социальной защиты (SSN), а во втором – идентификационным номером налогоплательщика (TIN). Частные лица имеют номера социальной защиты, а предприятия – номера налогоплательщиков:

```
public interface CustomerHomeRemote extends javax.ejb.EJBHome {

    public CustomerRemote createWithSSN(Integer id, String
    socialSecurityNumber)
        throws CreateException, RemoteException;

    public CustomerRemote createWithTIN(Integer id, String
    taxIdentificationNumber)
        throws CreateException, RemoteException;

    public CustomerRemote findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;
}
```

Суффиксы полезны, если требуется, чтобы имена методов `create()` были более содержательными, или необходимо сузить их специализацию при переопределении методов. У каждого метода `create<`*`СУФФИКС`*`>()` должен быть соответствующий ему метод `ejbCreate<`*`СУФФИКС`*`>()` в классе компонента. Например, в классе `CustomerBean` требуется определить методы `ejbCreateWithSSN()` и `ejbCreateWithTIN()`, а также соответствующие им методы `ejbPostCreateWithSSN()` и `ejbPostCreateWithTIN()`. Но мы хотим, чтобы этот пример оставался простым, поэтому обойдемся одним методом `create()` без суффиксов.

Enterprise JavaBeans указывает, что метод `create()` в удаленном внутреннем интерфейсе должен генерировать исключение `javax.ejb.CreateException`. В случае с постоянством, управляемом контейнером, контейнеру необходимо общее исключение для обнаружения коммуникационных ошибок во время процесса создания компонента.

Удаленные внутренние интерфейсы объектов должны определять метод `findByPrimaryKey()`, принимающий первичный ключ компонента в

качестве единственного аргумента, но не надо определять никаких соответствующих ему методов в классе компонента. Реализация этого метода создается автоматически средством развертывания. Во время выполнения метод `findByPrimaryKey()` будет автоматически находить и возвращать удаленную ссылку на объектный компонент с соответствующим первичным ключом.

Разработчик компонента может также объявить другие поисковые методы. Например, удаленный интерфейс компонента Customer мог бы определить метод `findByLastName()`, ищущий все объекты Customer с указанной фамилией. Такие типы поисковых методов автоматически реализуются средствами развертывания с использованием сигнатуры методов и оператора языка EJB QL, похожего на SQL, но специфичного для EJB. Пользовательские поисковые методы и язык EJB QL детально обсуждаются в главе 8.

XML-дескриптор развертывания

Объектные компоненты CMP 2.0 должны быть упакованы для развертывания с помощью XML-дескриптора развертывания, описывающего компонент и его абстрактную схему постоянства. В большинстве случаев разработчик компонента не создает дескриптор развертывания непосредственно, а чаще прибегает к визуальным средствам развертывания контейнера для упаковки компонентов. В этой книге, однако, я буду детально описывать объявления в дескрипторе развертывания, поэтому вы получите полное представление об их содержании и структуре.

Для нашего простого компонента Customer дескриптор развертывания содержит много элементов, которые должны быть вам знакомы по главе 4. Здесь же наиболее важными для нас будут элементы, специфичные для объектных компонентов и постоянства. Далее приведен полный дескриптор развертывания для компонента Customer:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
            <ejb-class>com.titan.customer.CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Customer</abstract-schema-name>
            <cmp-field><field-name>id</field-name></cmp-field>
```

```
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <role-name>Employees</role-name>
        </security-role>
        <method-permission>
            <role-name>Employees</role-name>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        <container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Первые несколько элементов, объявляющие имя компонента (CustomerEJB), а также внутренний, удаленный классы и класс компонента, должны быть уже вам знакомы. Элемент <security-identity> обсуждался в главе 3.

Элементы <assembly-descriptor>, задающие значения атрибутов транзакций и безопасности компонента, были также кратко рассмотрены в главе 4. В данном случае они утверждают, что все работники могут получить доступ к любым методам компонента CustomerEJB и что все методы используют атрибут транзакции Required.

Кроме этого для объектов с постоянством, управляемым контейнером, нужно задать их тип постоянства, версию и указать, допускают ли они повторное вхождение – реентерабельны ли (reentrant) они. Эти элементы объявляются внутри элемента <entity>.

Элемент <persistence-type> сообщает системе контейнера, является ли компонент объектом с постоянством, управляемым контейнером, или постоянство реализуется им самостоятельно. В нашем случае постоянство управляется контейнером, поэтому мы использовали значение Container. Если бы оно реализовывалось компонентом, значение было бы Bean.

Элемент <cmp-version> сообщает системе контейнера, какая версия постоянства, управляемого контейнером, используется. Контейнеры

EJB 2.0 должны поддерживать новую модель постоянства, управляемого контейнером, вместе со старой моделью, определенной в EJB 1.1. Значение элемента <cmp-version> должно быть либо 2.x, либо 1.x для версий EJB 2.0 и 1.0, соответственно. Элемент <cmp-version> не обязателен. Если он не задан, принимается значение 2.x. В нашем случае он не нужен и указан в качестве подсказки для других разработчиков, которые могут увидеть данный дескриптор.

Элемент <reentrant> указывает, допустимо ли реентерабельное поведение. В нашем случае он имеет значение False, указывающее, что компонент не реентерабельный (т. е. не допускающий циклических вхождений). Значение True означало бы то, что компонент CustomerEJB реентерабельный и повторные вхождения разрешены. Реентерабельность обсуждалась в главе 3.

Объектный компонент должен также объявить свои поля постоянства, управляемого контейнером, и свой первичный ключ:

```
<entity>
    <ejb-name>CustomerEJB</ejb-name>
    <home>com.titan.customer.CustomerHomeRemote</home>
    <remote>com.titan.customer.CustomerRemote</remote>
    <ejb-class>com.titan.customer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>lastName</field-name></cmp-field>
    <cmp-field><field-name>firstName</field-name></cmp-field>
    <primkey-field>id</primkey-field>
</entity>
```

Полями постоянства, управляемого контейнером, являются id, lastName и firstName, как указано в элементе <cmp-field>. Элементы <cmp-field> должны иметь соответствующие им методы доступа в классе CustomerBean. Как видно из табл. 6.1, значения, объявленные в элементе <field-name>, соответствуют именам абстрактных методов доступа, объявленных нами в классе CustomerBean. Во время соотнесения методов с объявлениями в элементе <field-name> приставки get- и set- в именах методов игнорируются.

Таблица 6.1. Имена полей для абстрактных методов доступа

Поле CMP	Абстрактный метод доступа
id	public abstract Integer getId() public abstract void setId(Integer id)
lastName	public abstract String getLastName() public abstract void setLastName(String lname)
firstName	public abstract String getFirstName() public abstract void setFirstName(String fname)

CMP 2.0 требует, чтобы значение элемента <field-name> начиналось со строчной буквы, тогда как соответствующий ему метод должен быть записан в виде get<значение field-name>(), set<значение field-name>(), и первая буква значения <field-name> должна быть преобразована в прописную. Тип возвращаемого значения метода get и параметра метода set определяется типом <cmp-field>. Соглашение о том, что имена полей, состоящие из нескольких слов, объявляются с использованием «переменного регистра», в котором каждое новое слово начинается с заглавной буквы (например, lastName), принято в данной книге и не является требованием CMP 2.0.

Наконец, мы объявляем первичный ключ, используя два поля: <prim-key-class> и <primkey-field>. Элемент <prim-key-class> указывает тип первичного ключа, а <primkey-field> указывает поле, предназначенное для первичного ключа. В компоненте Customer применяется *простой* (*single-field*) первичный ключ, в котором только одно управляемое контейнером поле описывает уникальный идентификатор компонента. Элемент <primkey-field> должен быть объявлен в том случае, когда объектный компонент использует простой первичный ключ. Вместо него часто используются *составные* (*compound*) первичные ключи, содержащие несколько постоянных полей. В этом случае разработчик компонента создает пользовательский первичный ключ. Элемент <prim-key-class> требуется всегда, независимо от того, является ли первичный ключ простым, составным или неизвестным. Поле в *неизвестном* (*unknown*) ключе вовсе не может быть объявлено в компоненте. Различные типы первичных ключей детально рассмотрены в главе 11.

Файл EJB JAR

Итак, мы создали интерфейсы, класс компонента и дескриптор развертывания и готовы упаковать свой компонент для развертывания. Как нам известно из главы 4, файл JAR предоставляет способ «совместного архивирования» компонента, обеспечивающий монолитность последнего и возможность его развертывания в контейнере EJB. Примеры, доступные на *http://www.orielly.com/catalog/entjbeans3/*, содержат должным образом подготовленный файл JAR, включающий интерфейсы, класс и дескриптор развертывания компонента Customer. Можно воспользоваться этими файлами или создать их вручную. Команда, необходимая для создания нового файла EJB JAR:

```
\dev % jar cf customer.jar com/titan/customer/*.class
com/titan/customer/META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\customer\*.class com\titan\customer
\META-INF\ejb-jar.xml
```

Большинство серверов EJB предоставляют графические средства либо средства командной строки для автоматического создания дескриптора развертывания и упаковки компонента в архив JAR. Некоторые из

этих инструментов автоматически создают даже внутренние и удаленные интерфейсы на основе сведений, полученных от разработчика. Те, кто предпочитает такие средства, могут обратиться к рабочим книгам и пройти процесс развертывания объектного компонента с помощью специфичных для каждого производителя средств развертывания.

Развертывание

После того как компонент Customer упакован в файл JAR, он уже готов к обработке средствами развертывания. У большинства производителей эти инструменты собраны в один графический пользовательский интерфейс, используемый во время развертывания. Их целью является связывание полей постоянства компонента с полями или объектами данных в базе данных. (Ранее в этой главе на рис. 6.2 и 6.3 были показаны два визуальных инструмента, применяемых для связывания полей постоянства компонента Customer.)

Кроме того, роли безопасности следует сопоставить с объектами в зоне безопасности целевого окружения, а также необходимо зарегистрировать компонент в службе имен и присвоить ему имя, используемое для его поиска (связать имя). Эти задачи выполняются также с применением средств развертывания, предоставляемых производителем. Рабочие книги содержат пошаговые инструкции по развертыванию компонента Customer в специфичных для каждого производителя средах.

Клиентское приложение

Приложение `Client` представляет собой удаленное клиентское приложение для компонента Customer, которое создаст несколько покупателей, найдет их, а затем удалит. Здесь приведено полное определение приложения `Client`.

```
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args)) throws Exception {
        // получаем CustomerHome
        Context jndiContext = getInitialContext();
        Object obj=jndiContext.lookup("CustomerHomeRemote");
        CustomerHomeRemote home = (CustomerHomeRemote)
         javax.rmi.PortableRemoteObject.narrow(obj,CustomerHomeRemote.class);
        // создаем компоненты Customer
        for(int i =0;i <args.length;i++){
            Integer primaryKey =new Integer(args [ i ]);
            String firstName = args [ i ];
            String lastName = args [ i ];
```

```
            CustomerRemote customer = home.create(primaryKey);
            customer.setFirstName(firstName);
            customer.setLastName(lastName);
        }
        // ищем и удаляем компоненты Customer
        for(int i =0;i <args.length;i++){
            Integer primaryKey = new Integer(args [i ]);
            CustomerRemote customer = home.findByPrimaryKey(primaryKey);
            String lastName = customer.getLastName();
            String firstName = customer.getFirstName();
            System.out.print(primaryKey+"=");
            System.out.println(firstName+" "+lastName);
            // удаляем Customer
            customer.remove();
        }
    }
    public static Context getInitialContext(
        throws javax.naming.NamingException {
        Properties p =new Properties();
        // ...задаем свойства JNDI, специфичные для производителя
        return new javax.naming.InitialContext(p);
    }
}
```

Приложение `Client` создает несколько компонентов Customer, устанавливает их имя и фамилию, распечатывает значения полей постоянства, после чего удаляет объекты из контейнерной системы и, в результате, из базы данных.

📖 Рабочее упражнение 6.1. Основы постоянства в CMP 2.0

Поля постоянства

Поля постоянства, управляемого контейнером (CMP), – это виртуальные поля, значения которых берутся прямо из базы данных. Постоянные поля могут быть сериализуемыми (serializable) или примитивными типами Java.

Сериализуемым типом Java может быть любой класс, реализующий интерфейс `java.io.Serializable`. Большинство инструментов развертывания легко обрабатывают типы `java.lang.String`, `java.util.Date` и «обертки» примитивов (`Byte`, `Boolean`, `Short`, `Integer`, `Long`, `Double` и `Float`), т. к. эти типы объектов являются частью ядра Java и естественно соотносятся с полями в реляционных и других базах данных.

Компонент Customer объявляет три сериализуемых поля – `id`, `lastName` и `firstName`, которые естественно соотносятся с полями `INT` и `CHAR` таблицы `CUSTOMER` базы данных.

Вы также можете определять свои собственные сериализуемые типы, называемые *классами зависимых значений* (*dependent value classes*),

и объявить их в качестве полей CMP. Однако если нет особой необходимости, я не рекомендовал бы применять эти пользовательские сериализуемые объекты в качестве типов постоянных полей – они обычно рекомендуются для неструктурированных типов, таких как мультимедиа-данные (изображения, бинарные объекты и т. д.). Произвольные классы зависимых значений, как правило, не будут естественно соответствовать типам базы данных, поэтому они должны храниться в сериализованном виде в некотором типе бинарного поля базы данных.

Сериализуемые объекты всегда возвращаются как копии, а не как ссылки, поэтому изменение сериализуемого объекта никак не повлияет на значение в базе данных. Значение такого объекта должно обновляться с помощью абстрактного метода `set<имя-поля>`.

Примитивные типы (`byte`, `short`, `int`, `long`, `double`, `float` и `boolean`) также могут применяться в качестве полей CMP. Эти типы легко проецируются на базу данных и поддерживаются всеми инструментами развертывания. В качестве примера компонент Customer может объявить поле `boolean`, представляющее значение платежеспособности клиента.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }

    // Абстрактные методы доступа

    public abstract boolean getHasGoodCredit();
    public abstract void setHasGoodCredit(boolean creditRating);
```

Классы зависимых значений

В предыдущем разделе мы говорили, что классы зависимых значений представляют собой пользовательские сериализуемые объекты, которые могут выступать в качестве постоянных полей (хотя их применение и не рекомендуется). Классы зависимых значений полезны для упаковки данных и перемещения их между объектными компонентами и удаленными клиентами. Они отделяют внешний вид компонента со стороны клиента от абстрактной модели постоянства, что упрощает изменение класса объектного компонента без влияния на существующий клиент.

Методы удаленного интерфейса объектного компонента следует определять независимо от возможной абстрактной схемы постоянства. Другими словами, следует разрабатывать удаленные интерфейсы для моделирования прикладных понятий, а не лежащей в их основе программной модели постоянства. Классы зависимых значений помогают отделить представление для удаленного клиента от модели постоян-

ства с помощью объектов, заполняющих пробелы в этой схеме. Классы зависимых значений в основном применяются в удаленных интерфейсах, где упаковка данных может уменьшить загрузку сети. В локальных интерфейсах они менее полезны.

Например, компонент Customer мог бы быть изменен таким образом, чтобы его поля `lastName` и `firstName` не выставлялись удаленным клиентам через методы доступа. Это разумный подход к разработке, т. к. большинство клиентов запрашивают имя сразу целиком. В этом случае удаленный интерфейс может быть изменен следующим образом:

```
import java.rmi.RemoteException;

public interface CustomerRemote extends javax.ejb.EJBObject {

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

}
```

Этот удаленный интерфейс проще, чем тот, который мы видели раньше. Он позволяет удаленным клиентам получить всю информацию об имени за один вызов метода вместо двух. Это уменьшает загрузку сети и увеличивает производительность удаленных клиентов. Кроме того, такое использование зависимого значения `Name` лучше соответствует имеющему место взаимодействию клиентов с компонентом Customer в смысле семантики.

Для реализации этого интерфейса к классу `CustomerBean` добавлен прикладной метод, соответствующий методам удаленного интерфейса. Метод `setName()` обновляет поля `lastName` и `firstName`, а метод `getName()` создает из этих полей объект `Name`:

```
import javax.ejb.EntityContext;

public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id) {
    }
    // Прикладные методы
    public Name getName() {
        Name name = new Name(getLastName(),getFirstName());
        return name;
    }
    public void setName(Name name) {
        setLastName(name.getLastName());
        setFirstName(name.getFirstName());
    }
```

```
    // Абстрактные методы доступа

    public abstract String getLastName();
    public abstract void setLastName(String lname);

    public abstract String getFirstName();
    public abstract void setFirstName(String fname);
```

Это хороший пример применения классов зависимых значений для отделения клиентского представления от абстрактной схемы постоянства.

Методы `getName()` и `setName()` не являются абстрактными методами постоянства, это прикладные методы. У объектных компонентов может быть столько прикладных методов, сколько необходимо. Прикладные методы вкладывают в компонент Customer прикладную логику, в противном случае компонент был бы только «оберткой» для данных. Например, в метод `setName()` может быть добавлена проверка, гарантирующая правильность данных перед выполнением обновления. Кроме того, класс компонента может использовать другие методы, помогающие обрабатывать данные, – они являются методами экземпляра и не могут выставляться в удаленном интерфейсе в качестве прикладных методов.

Способ, которым определяются классы зависимых значений, важен для понимания того, как они должны применяться. Класс зависимого значения Name определяется следующим образом:

```
public class Name implements java.io.Serializable {
    private String lastName;
    private String firstName;

    public Name(String lname, String fname){
        lastName = lname;
        firstName = fname;
    }
    public String getLastName() {
        return lastName;
    }
    public String getFirstName() {
        return firstName;
    }
}
```

Вы видите, что у класса зависимого значения Name есть методы доступа get, но нет методов set. И это неизменно. Это стратегия разработки, используемая в книге, а не требование спецификации. CMP 2.0 не указывает, как должны определяться классы зависимых значений. Мы сделали зависимые значения неизменяемыми, поэтому удаленные клиенты не смогут изменить поле `Name` объекта. Причина проста. Объект `Name` представляет собой копию, а не удаленную ссылку. Изменение объекта `Name` не отразится на базе данных. Объявление объекта неизменным гарантирует, что клиенты не примут это зависимое значе-

ние за ссылку на удаленный объект, думая, что изменения в объекте Name автоматически отразятся на базе данных. Для того чтобы изменить имя объекта, клиенту необходимо создать новый объект Name и вызвать метод setName() для обновления компонента Customer.

Следующий код иллюстрирует, как клиент должен модифицировать имя объекта с помощью класса зависимого значения Name:

```
// Ищем компонент Customer
customer = home.findByPrimaryKey(primaryKey);
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName()+" "+name.getLastName());

// Изменяем имя компонента Customer
name = new Name("Monson-Haefel", "Richard");
customer.setName(name);
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName()+" "+name.getLastName());
```

Результат будет выглядеть так:

```
1 = Richard Monson
1 = Richard Monson-Haefel
```

Определение интерфейсов компонентов в соответствии с прикладными понятиями, а не лежащими в их основе данными не всегда удобно, но вам следует, насколько это возможно, придерживаться такой стратегии в тех случаях, когда модель базовых данных не очевидно соответствует прикладным целям или понятиям, моделируемым объектным компонентом. Интерфейсы компонента могут применяться разработчиками, знающими прикладную область, но не знакомыми с абстрактной моделью программирования. Для них важно, как объектный компонент отражает прикладное понятие. Кроме того, определение интерфейсов независимо от модели постоянства делает возможным отдельное развитие интерфейсов компонента и модели постоянства. Благодаря этому абстрактную программную модель постоянства при необходимости можно изменить в любое время, а поведение компонента сделать расширенным.

Классы зависимых значений полезны, но использовать их следует разумно. Вообще говоря, глупо применять классы зависимых значений, когда поля CMP могут работать так же хорошо. Например, проверка платежеспособности покупателя перед обработкой заказа может быть легко выполнена с помощью прямого вызова метода getHasGoodCredit(). В этом случае класс зависимого значения не нужен.

Рабочее упражнение 6.2. Классы зависимых значений CMP 2.0

Поля отношений

Объектные компоненты могут формировать отношения с другими объектными компонентами. На рис. 6.1 был показан компонент Customer, имеющий отношение типа «один-к-одному» с компонентом Address. Компонент Address – это небольшой прикладной объект, к которому всегда нужно обращаться в контексте другого объектного компонента. Это означает, что у него могут быть только локальные интерфейсы и не должно быть удаленных. Объектный компонент в одно и то же время может поддерживать отношения с несколькими разными объектными компонентами. Например, мы без труда могли бы добавить поля отношения для объектных компонентов Phone, CreditCard и других, связанных с компонентом Customer. Однако пока мы не хотим усложнять компонент Customer.

Руководствуясь рис. 6.1, определим компонент Address следующим образом:

```
package com.titan.address;

import javax.ejb.EntityContext;

public abstract class AddressBean implements javax.ejb.EntityBean {

    public Integer ejbCreateAddress(String street, String city,
        String state, String zip)
    {
         setStreet(street);
         setCity(city);
         setState(state);
         setZip(zip);
         return null;
    }
    public void ejbPostCreateAddress(String street, String city,
        String state, String zip) {
    }

    // Поля постоянства

    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract String getStreet();
    public abstract void setStreet(String street);
    public abstract String getCity();
    public abstract void setCity(String city);
    public abstract String getState();
    public abstract void setState(String state);
    public abstract String getZip();
    public abstract void setZip(String zip);

    // Стандартные методы обратного вызова
```

```
    public void setEntityContext(EntityContext ec){}
    public void unsetEntityContext(){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
}
```

В классе `AddressBean` определены метод `ejbCreateAddress()`, вызываемый во время создания нового компонента Address, а также несколько полей постоянства (`street`, `city`, `state` и `zip`). Поля постоянства представлены абстрактными методами доступами, являющимися средством, требуемым для полей постоянства во всех классах объектных компонентов. Эти абстрактные методы доступа соответствуют относящемуся к ним набору XML-элементов дескриптора развертывания, определяющих абстрактную схему постоянства для компонента Address. Во время развертывания средство развертывания контейнера свяжет поля постоянства компонентов Customer и Address с базой данных. Это означает, что в нашей реляционной базе данных должна существовать таблица, содержащая столбцы, которые соответствовали бы полям постоянства компонента Address. В нашем примере адресную информацию мы будем хранить в отдельной таблице `ADDRESS`, но данные могли бы быть связаны и с любой другой таблицей:

```
CREATE TABLE ADDRESS
(
    ID INT PRIMARY KEY NOT NULL,
    STREET CHAR(40),
    CITY CHAR(20),
    STATE CHAR(2),
    ZIP CHAR(10)
)
```

В объектных компонентах не требуется определять в качестве полей постоянства все столбцы из соответствующих им таблиц. В действительности объектный компонент может даже иметь не одну соответствующую ему таблицу; он может сохраняться в нескольких таблицах. Основной момент здесь заключается в том, что средство развертывания контейнера позволяет отображать абстрактную схему постоянства объектных компонентов на базу данных различными способами, проводя явное разграничение между классами постоянства и базой данных. Столбец `ID` представляет собой автоинкрементное поле, автоматически создаваемое базой данных или контейнерной системой. Это первичный ключ компонента Address.

После создания компонента его первичный ключ больше не может модифицироваться. Если первичные ключи являются автогенерируемыми значениями, такими как столбец `ID` в таблице `ADDRESS`, контейнер EJB будет извлекать значение первичного ключа из базы данных.

В дополнение к классу компонента Address мы определим для него локальный интерфейс, позволяющий обращаться к нему другим объектным компонентам (а именно, компоненту Customer) внутри одного адресного пространства или процесса:

```
// Локальный интерфейс компонента Address
public interface AddressLocal extends javax.ejb.EJBLocalObject {
    public String getStreet();
    public void setStreet(String street);
    public String getCity();
    public void setCity(String city);
    public String getState();
    public void setState(String state);
    public String getZip();
    public void setZip(String zip);
}

// Локальный внутренний интерфейс компонента Address
public interface AddressHomeLocal extends javax.ejb.EJBLocalHome {
    public AddressLocal createAddress(String street,String city,
        String state,String zip) throws javax.ejb.CreateException;
    public AddressLocal findByPrimaryKey(Integer primaryKey)
        throws javax.ejb.FinderException;

}
```

Читатели, возможно, заметили, что и в методе `ejbCreate()` класса `AddressBean`, и в методе `findByPrimaryKey()` внутреннего интерфейса тип первичного ключа определен как `java.lang.Integer`. Как я упоминал ранее, первичный ключ является автогенерируемым. Большинство производителей EJB 2.0 позволяют сопоставлять первичные ключи объектных компонентов с автогенерируемыми полями. Если сервер не поддерживает автогенерируемые первичные ключи, необходимо устанавливать значение первичного ключа в методе `ejbCreate()`. Это, как правило, справедливо для первичных ключей с единственным значением, но не обязательно для составных первичных ключей.

Поле отношения для компонента Address определено в классе `CustomerBean` при помощи абстрактного метода доступа, таким же образом, каким описываются и поля постоянства. В следующем коде `CustomerBean` был изменен с целью включения в качестве поля отношения компонента Address:

```
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...
    // Отношения постоянства
    public abstract AddressLocal getHomeAddress();
    public abstract void setHomeAddress(AddressLocal address);
    // Поля постоянства
```

```
public abstract boolean getHasGoodCredit();
public abstract void setHasGoodCredit(boolean creditRating);
...
```

Методы доступа `getHomeAddress()` и `setHomeAddress()` достаточно очевидны. Они позволяют обращаться к компоненту и изменять его отношение `homeAddress`. Эти методы доступа представляют *поле отношения* (*relationship field*) – виртуальное поле, ссылающееся на другой объектный компонент. Имя метода доступа определяется именем поля отношения, объявленного в XML-дескрипторе развертывания. Мы назвали адрес клиента `homeAddress`, поэтому соответствующие ему имена методов доступа будут `getHomeAddress()` и `setHomeAddress()`.

Для того чтобы установить отношение между компонентом Customer и его собственным адресом, к таблице `CUSTOMER` будет добавлен внешний ключ `ADDRESS_ID`. Внешний ключ будет указывать на запись в таблице `ADDRESS`. Эта схема на самом деле практически обратна по отношению к той, которая обычно используется, когда таблица `ADDRESS` содержит внешний ключ, ссылающийся на таблицу `CUSTOMER`. Однако применяемая здесь схема полезна для демонстрации альтернативных отображений баз данных и еще встретится нам в главе 7:

```
CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY NOT NULL,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
    ADDRESS_ID INT
)
```

Когда создается новый компонент Address и устанавливается как отношение `homeAddress` компонента Customer, первичный ключ компонента Address помещается в столбец `ADDRESS_ID` таблицы `CUSTOMER`, и формируется отношение в базе данных:

```
// Получаем локальную ссылку
AddressLocal address = ...

// Устанавливаем отношение
customer.setHomeAddress(address);
```

Для того чтобы присвоить компоненту Customer домашний адрес, нам потребуется передать ему адресную информацию. На первый взгляд кажется, что с этой целью достаточно объявить в удаленном интерфейсе соответствующие методы доступа `setHomeAddress()` и `getHomeAddress()`, но это не так! Хотя поля постоянства можно сделать непосредственно доступными удаленным клиентам, отношения постоянства более сложны.

Поля отношения компонента не разрешается делать доступными через его удаленный интерфейс. В случае с полем `homeAddress` мы объявили,

что его тип будет AddressLocal, который представляет собой локальный интерфейс, поэтому методы доступа setHomeAddress() и getHomeAddress() не могут быть объявлены в удаленном интерфейсе компонента Customer. Причина такого ограничения, наложенного на удаленные интерфейсы, довольно проста: EJBLocalObject, реализующий локальный интерфейс, оптимизирован для применения внутри этого же адресного пространства или процесса, в котором находится экземпляр компонента, и не может использоваться из любой точки сети. Другими словами, ссылки, реализующие локальный интерфейс компонента, не могут передаваться через сеть, поэтому локальный интерфейс не может быть объявлен в качестве типа возвращаемого значения параметра удаленного интерфейса.

Мы эксплуатируем достоинства оптимизации EJBLocalObject для улучшения производительности, но это же самое преимущество ограничивает прозрачность расположения. Мы должны использовать ее только в пределах того же адресного пространства.

С другой стороны, локальные интерфейсы (интерфейсы, расширяющие javax.ejb.EJBLocalObject) могут выставлять любой вид полей отношений. В случае применения локальных интерфейсов вызывающий объект и вызываемый компонент расположены в одном адресном пространстве, поэтому они могут передавать друг другу локальные ссылки без каких-либо проблем. Например, если бы мы определили локальный интерфейс для компонента Customer, в нем мог бы содержаться метод, позволяющий локальным клиентам обращаться непосредственно к его отношению Address:

```
public interface CustomerLocal extends javax.ejb.EJBLocalObject {
    public AddressLocal getHomeAddress();
    public void setHomeAddress(AddressLocal address);
}
```

Что касается компонента Address, лучше определить для него локальный интерфейс только потому, что он представляет собой достаточно небольшой компонент. Для того чтобы обойти ограничения, связанные с применением удаленного интерфейса, прикладные методы в классе компонента обмениваются данными об адресе, а не ссылками на компонент Address. Например, мы можем объявить метод, позволяющий клиенту посылать адресную информацию для создания домашнего адреса для компонента Customer:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id) {
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id) {
    }
```

```
    // Прикладной метод
    public void setAddress(String street,String city,String state,String zip)
{
        try {

            AddressLocal addr = this.getHomeAddress();
            if(addr == null) {
                // У Customer еще нет адреса. Создаем его.
                InitialContext cntx = new InitialContext();
                AddressHomeLocal addrHome =
                    (AddressHomeLocal)cntx.lookup("AddressHomeLocal");
                addr = addrHome.createAddress(street,city,state,zip);
                this.setHomeAddress(addr);
            } else {
                // У Customer уже есть адрес. Меняем его поля.
                addr.setStreet(street);
                addr.setCity(city);
                addr.setState(state);
                addr.setZip(zip);
            }

        } catch(Exception e) {
            throw new EJBException(e);
        }
    }

    ...
```

В удаленном интерфейсе компонента Customer также объявлен прикладной метод `setAddress()` класса `CustomerBean`, поэтому он может вызываться удаленными клиентами:

```
public interface Customer extends javax.ejb.EJBObject {

  public void setAddress(String street,String city,String state,String zip);

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

    public boolean getHasGoodCredit() throws RemoteException;
    public void setHasGoodCredit(boolean creditRating) throws
RemoteException;

}
```

Параметры вызываемого прикладного метода `CustomerRemote.setAddress()` класса `CustomerBean` используются для создания нового компонента Address и установки его в качестве поля отношения `homeAddress`, если оно еще не существует. Если компонент Customer уже имеет отношение `homeAddress`, то компонент Address модифицируется таким образом, чтобы он отражал новую информацию об адресе.

При создании нового компонента Address из контекста имен окружения (Environment Naming Context, ENC) JNDI извлекается внутрен-

ний объект и вызывается его метод `createAddress()`. Это приводит к созданию нового компонента Address и вставке в базу данных соответствующей записи `ADDRESS`. Созданный компонент Address используется в методе `setHomeAddress()`. Класс `CustomerBean` должен явно вызвать метод `setHomeAddress()`, иначе новый адрес не будет связан с клиентом. Фактически просто создание компонента Address без связывания его с клиентом с помощью метода `setHomeAddress()` приведет к *разъединенному* (*disconnected*) компоненту Address. Более точно, это приведет к появлению в базе данных записи `ADDRESS`, которая не ссылается ни на какую запись в таблице `CUSTOMER`. Во многих случаях применение разъединенных объектных компонентов абсолютно нормально, а иногда даже желательно. Однако в данном случае мы хотим, чтобы новый компонент Address был связан с полем отношения `homeAddress` компонента Customer.

Жизнеспособность разъединенных объектов зависит частично от ссылочной целостности базы данных. Например, если ссылочная целостность разрешает только ненулевые значения столбца внешнего ключа, создание разъединенного объекта способно вызвать ошибку базы данных.

При вызове метода `setHomeAddress()` контейнер автоматически связывает запись в таблице `ADDRESS` с записью в `CUSTOMER`. В этом случае он заносит первичный ключ таблицы `ADDRESS` в поле `ADDRESS_ID` записи `CUSTOMER` и создает ссылку в записи `CUSTOMER` на запись `ADDRESS`.

Если у компонента Customer уже есть `homeAddress`, нам необходимо вместо создания нового компонента Address просто изменить его значение. Нам не требуется использовать `setHomeAddress()`, если мы просто модифицируем значения существующего компонента Address, поскольку компонент Address, уже измененный нами, имеет отношение с этим объектным компонентом.

Также мы хотим дать клиенту прикладной метод для получения информации о домашнем адресе компонента Customer. Поскольку мы не можем послать экземпляр компонента Address напрямую клиенту (ведь это локальный интерфейс), мы должны упаковать сведения об адресе в некоторую другую форму и переслать их клиенту. Существуют два решения этой проблемы: получение и возвращение удаленного интерфейса компонента Address и возвращение данных в виде объекта зависимого значения.

Мы можем получить удаленный интерфейс компонента Address, только если он был определен. Объектные компоненты могут иметь набор или локальных интерфейсов, или удаленных интерфейсов, или и тех и других. В ситуации, с которой мы имеем дело, компонент Address слишком мал, чтобы оправдать создание для него удаленного интерфейса, но во многих других случаях компоненту действительно может

понадобиться удаленный интерфейс. Если, например, компонент Customer ссылается на компонент SalesPerson, CustomerBean может конвертировать локальную ссылку в удаленную. Это может быть достигнуто путем обращения к локальному компонентному объекту, получения его первичного ключа (EJBLocalObject.getPrimaryKey()), получения из JNDI ENC удаленного внутреннего интерфейса компонента SalesPerson и последующего использования первичного ключа и удаленной внутренней ссылки для поиска ссылки на удаленный интерфейс:

```
public SalesRemote getSalesRep(){
    SalesLocal local = getSalesPerson();
    Integer primKey = local.getPrimaryKey();

    Object ref = jndiEnc.lookup("SalesHomeRemote");
    SalesHomeRemote home = (SalesHomeRemote)
        PortableRemoteObject.narrow(ref, SalesHomeRemote.class);

    SalesRemote remote = home.findByPrimaryKey( primKey );
    return remote;
}
```

Другая возможность для передачи данных компонента Address между удаленными клиентами и компонентом Customer заключается в использовании зависимого значения. Этот подход рекомендуется для небольших компонентов, таких как компонент Address. Вообще, мы не должны непосредственно выставлять эти компоненты удаленным клиентам.

В следующем коде показано, как вместе с локальными компонентными интерфейсами компонента Address применяется класс зависимого значения AddressDO (окончание DO в имени AddressDO является соглашением, принятым в этой книге, это сокращение от «dependent object» – «зависимый объект»):

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id) {
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id) {
    }
    // Прикладной метод
    public AddressDO getAddress() {
        AddressLocal addrLocal = getHomeAddress();
        if(addrLocal == null) return null;
        String street = addrLocal.getStreet();
        String city = addrLocal.getCity();
        String state = addrLocal.getState();
        String zip = addrLocal.getZip();
        AddressDO addrValue = new AddressDO(street,city,state,zip);
```

```
        return addrValue;
    }
    public void setAddress(AddressDO addrValue)
        throws EJBException {

        String street = addrValue.getStreet();
        String city = addrValue.getCity();
        String state = addrValue.getState();
        String zip = addrValue.getZip();

        AddressLocal addr = getHomeAddress();

        try {

        if(addr == null) {
            // У Customer еще нет адреса. Создаем его.
            InitialContext cntx = new InitialContext();
            AddressHomeLocal addrHome = (AddressHomeLocal)cntx.lookup
                ("AddressHomeLocal");
            addr = addrHome.createAddress(street, city, state, zip);
            this.setHomeAddress(addr);
        } else {
            // У Customer уже есть адрес. Изменяем его поля.
            addr.setStreet(street);
            addr.setCity(city);
            addr.setState(state);
            addr.setZip(zip);
        }

        } catch(NamingException ne) {
            throw new EJBException(ne);
        } catch(CreateException ce) {
            throw new EJBException(ce);
        }

    }
    ...
```

Далее приводится определение для класса зависимого значения `AddressDO`**, используемого компонентом для отправки клиенту информации об адресе:**

```
public class AddressDO implements java.io.Serializable {
    private String street;
    private String city;
    private String state;
    private String zip;

    public AddressDO(String street, String city, String state, String zip ) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
```

```
    }
    public String getStreet() {
        return street;
    }
    public String getCity() {
        return city;
    }
    public String getState() {
        return state;
    }
    public String getZip() {
        return zip;
    }
}
```

Зависимое значение `AddressDO` следует соглашениям, принятым в этой книге. Оно неизменно. Это означает, что после своего создания оно не может быть изменено. Как было указано выше, неизменность помогает подкрепить тот факт, что класс зависимого значения представляет собой копию, а не удаленную ссылку.

Теперь вы можете воспользоваться клиентским приложением для проверки отношения компонента Customer с компонентом Address. Здесь приведен код для клиента, который создает новый Customer, дает ему адрес, затем изменяет этот адрес, применяя метод, описанный выше:

```
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args) throws Exception {
        // Получаем CustomerHome
        Context jndiContext = getInitialContext();
        Object obj=jndiContext.lookup("CustomerHomeRemote");
        CustomerHome home = (CustomerHomeRemote)
            javax.rmi.PortableRemoteObject.narrow(obj,
            CustomerHomeRemote.class);

        // Создаем Customer
        Integer primaryKey = new Integer(1);
        Customer customer = home.create(primaryKey);

        // Создаем адрес
        AddressDO address = new AddressDO("1010 Colorado",
            "Austin", "TX", "78701");
        // Устанавливаем адрес
        customer.setAddress(address);

        address = customer.getAddress();
```

```
            System.out.print(primaryKey+" = ");
            System.out.println(address.getStreet());
            System.out.println(address.getCity()+","+
                               address.getState()+" "+
                               address.getZip());

            // Создаем новый адрес
            address = new AddressDO("1600 Pennsylvania Avenue NW",
                "DC", "WA", "20500");

            // Изменяем адрес компонента Customer
            customer.setAddress(address);

            address = customer.getAddress();

            System.out.print(primaryKey+" = ");
            System.out.println(address.getStreet());
            System.out.println(address.getCity()+","+
                               address.getState()+" "+
                               address.getZip());

            // Удаляем Customer
            customer.remove();
        }

        public static Context getInitialContext()
            throws javax.naming.NamingException {
            Properties p = new Properties();
            // ... задаем свойства JNDI, специфичные для производителя
            //return new javax.naming.InitialContext(p);
            return null;
        }
    }
```

В следующем листинге показан дескриптор развертывания для компонентов Customer и Address. Пока мы не должны вдаваться в детали дескриптора развертывания, который будет более подробно рассмотрен в главе 7.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
            <ejb-class>com.titan.customer.CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
```

```
            <abstract-schema-name>Customer</abstract-schema-name>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
        <entity>
            <ejb-name>AddressEJB</ejb-name>
            <local-home>com.titan.address.AddressHomeLocal</local-home>
            <local>com.titan.address.AddressLocal</local>
            <ejb-class>com.titan.address.AddressBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Address</abstract-schema-name>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>street</field-name></cmp-field>
            <cmp-field><field-name>city</field-name></cmp-field>
            <cmp-field><field-name>state</field-name></cmp-field>
            <cmp-field><field-name>zip</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
    </enterprise-beans>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Customer-Address</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Customer-has-an-Address
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>CustomerEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>homeAddress</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Address-belongs-to-Customer
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>AddressEJB</ejb-name>
                </relationship-role-source>
            </ejb-relationship-role>
        </ejb-relation>
```

```
    </relationships>
    <assembly-descriptor>
        <security-role>
            <role-name>Employees</role-name>
        </security-role>
        <method-permission>
            <role-name>Employees</role-name>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <method>
                <ejb-name>AddressEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
                <ejb-name>AddressEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

📖 Рабочее упражнение 6.3. Простое отношение в CMP 2.0

7

EJB 2.0 CMP: Отношения между объектами

В главе 6 мы рассмотрели основы постоянства EJB 2.0, управляемого контейнером. Ознакомились с информацией о полях постоянства, управляемого контейнером, и с основными полями отношений, управляемых контейнером. Здесь мы продолжим разработку компонента Customer и рассмотрим подробно каждое из семи возможных отношений, которые могут существовать между объектными компонентами.

Для того чтобы объектные компоненты могли моделировать реальные прикладные понятия, они должны быть способны к формированию сложных отношений друг с другом. Это было трудно выполнить с помощью реализованного в EJB 1.1 постоянства, управляемого контейнером, из-за упрощенной модели программирования. В EJB 1.1 у объектных компонентов могли быть поля постоянства, но не могло быть полей отношений.

В EJB 2.0 поля отношений могут моделировать сложные взаимоотношения между объектными компонентами. В главе 6 мы продемонстрировали отношение «один-к-одному» между компонентами Customer и Address. Эти отношения были однонаправленными. У Customer была ссылка на Address, а у Address не было обратной ссылки на Customer. Это допустимые отношения, но возможны также и некоторые другие. Например, каждый компонент Address также мог бы ссылаться на свой Customer. Это уже пример двунаправленных отношений «один-к-одному», в которых оба участника поддерживают ссылки друг на друга. Кроме отношений «один-к-одному» объектные компоненты мо-

гут состоять в отношениях «один-ко-многим», «многие-к-одному» и «многие-ко-многим». Например, компонент Customer может иметь несколько телефонных номеров, но каждый номер принадлежит только одному компоненту Customer (отношение «один-ко-многим»). Также возможно, что клиент в прошлом был в нескольких круизах, а в каждом круизе было много клиентов (отношения «многие-ко-многим»).

Семь типов отношений

Между компонентами могут существовать семь типов отношений. Данная глава посвящена исследованию этих отношений и взаимодействия кода компонентов и дескрипторов развертывания – для того чтобы описать эти отношения. Для начала посмотрим на допустимые типы отношений. Имеются четыре типа отношений по количеству элементов: «один-к-одному», «один-ко-многим», «многие-к-одному» и «многие-ко-многим». Вдобавок к этому, каждое отношение может быть как однонаправленным, так и двунаправленным. Это дает восемь возможных комбинаций, но, если вы поразмышляете над этим, то поймете, что двунаправленные отношения «один-ко-многим» и «многие-к-одному» фактически ничем не отличаются друг от друга. Таким образом, имеется лишь семь различных типов отношений. Разобраться с этими отношениями нам помогут несколько простых примеров. Мы подробно остановимся на следующих:

Однонаправленные «один-к-одному»

Отношения между клиентом и адресом. Ясно, что желательно иметь возможность найти адрес клиента, но вероятно, что искать клиента по его адресу не потребуется.

Двунаправленные «один-к-одному»

Отношения между клиентом и номером кредитной карточки. Регистрируя клиента, очевидно, желательно иметь возможность найти номер его кредитной карточки. Вероятно также, что, зарегистрировав номер кредитной карточки, вы захотите найти клиента, владеющего этой кредитной карточкой.

Однонаправленные «один-ко-многим»

Отношения между клиентом и его телефонным номером. У клиента может быть несколько номеров (рабочий, домашний, сотовый и т. д.). Впоследствии может понадобиться найти номер телефона клиента, но вероятно, что какой-то один из них для поиска клиента не понадобится.

Двунаправленные «один-ко-многим»

Отношения между круизом и резервированием билетов. Хотелось бы при наличии билета иметь возможность найти круиз, для которого билет был зарезервирован. А также возможность найти по

имеющемуся круизу все зарезервированные для него билеты. Обратите внимание, что двунаправленные отношения «многие-к-одному» представляют собой лишь взгляд под другим углом на то же самое понятие.

Однонаправленные «многие-к-одному»

Отношения между круизом и судном. Хотелось бы иметь возможность найти судно, предназначенное для проведения отдельного круиза, с учетом того, что это судно может участвовать в нескольких круизах, хотя и в разное время. Не так очевидна польза от возможности найти судно для того, чтобы посмотреть, какие круизы с ним связаны, но можно реализовать двунаправленные отношения «многие-к-одному» с целью резервирования оной.

Однонаправленные «многие-ко-многим»

Отношения между резервированием билетов и каютой. В случае резервирования сразу для нескольких кают нам понадобится возможность найти каюту, связанную с резервированием. Однако вряд ли потребуется искать резервирование, связанное с отдельной каютой. (Тот, кто думает, что ему это понадобится, должен реализовать двунаправленное отношение.)

Двунаправленные «многие-ко-многим»

Отношения между круизом и клиентом. Клиент может резервировать билеты на несколько круизов, а в каждом круизе будут участвовать много клиентов. Очевидно, что надо предусмотреть возможность поиска и круизов, на которые у клиента заказан билет, и клиентов, которые собираются на данный круиз.

Абстрактная схема постоянства

В главе 6 мы научились формировать основные отношения между объектными компонентами Customer и Address с использованием абстрактной программной модели. Но на самом деле абстрактная программная модель – только половина уравнения. Кроме объявления абстрактных методов доступа разработчик компонента должен описать множественность и направление отношений между объектами в дескрипторе развертывания компонента. Все это обрабатывается в разделе `<relationship>` XML-дескриптора развертывания. По мере рассмотрения каждого типа отношений в следующих разделах мы исследуем и абстрактную программную модель, и элементы XML. Цель этого раздела состоит в том, чтобы познакомить читателя с основными элементами, используемыми в XML-дескрипторе развертывания, и в том, чтобы лучше подготовиться к последующим разделам, посвященным отдельным типам отношений.

В этой книге мы постоянно ссылаемся на программные средства Java, применяемые для описания отношений, – в частности, абстрактные

методы доступа – как на *абстрактную программную модель* (*abstract programming model*). Говоря об элементах XML дескриптора развертывания, мы оперируем термином *«абстрактная схема постоянства»* (*abstract persistence schema*). В спецификации EJB 2.0 термин «абстрактная схема постоянства» фактически относится и к идиомам Java, и к элементам XML, но здесь для облегчения обсуждения эти понятия разделены.

Абстрактная схема постоянства объектного компонента определена в разделе `<relationships>` дескриптора развертывания XML этого компонента. Раздел `<relationships>` разбит на разделы `<enterprise-beans>` и `<assembly-descriptor>`. Внутри элемента `<relationships>` каждое отношение типа «объект-объект» определено в отдельном элементе `<ejb-relation>`:

```
<ejb-jar>
    <enterprise-beans>
    ...
    </enterprise-beans>
    <relationships>
        <ejb-relation>
        ...
        </ejb-relation>
        <ejb-relation>
        ...
        </ejb-relation>
    </relationships>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

Определение полей отношений требует, чтобы для каждого отношения типа «объект-объект» в XML-дескриптор развертывания был добавлен элемент `<ejb-relation>`. Элементы `<ejb-relation>` дополняют абстрактную программную модель. Для каждой пары абстрактных методов доступа, определяющих поле отношений, в дескрипторе развертывания существует элемент `<ejb-relation>`. EJB 2.0 требует, чтобы объектные компоненты, участвующие в отношениях, были определены в том же самом XML-дескрипторе развертывания.

Здесь приведен фрагмент листинга дескриптора развертывания для компонентов Customer и Address, при этом элементы, определяющие отношения, выделены:

```
<ejb-jar>
    ...
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <local-home>com.titan.customer.CusomterHomeLocal</local-home>
            <local>com.titan.customer.CustomerLocal</local>
```

```
            ...
        </entity>
        <entity>
            <ejb-name>AddressEJB</ejb-name>
            <local-home>com.titan.address.AddressHomeLocal</local-home>
            <local>com.titan.address.AddressLocal</local>
            ...
        </entity>
        ...
    </enterprise-beans>

    <relationships>
        <ejb-relation>
            <ejb-relation-name>Customer-Address</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Customer-has-an-Address
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>CustomerEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>homeAddress</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Address-belongs-to-Customer
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>AddressEJB</ejb-name>
                </relationship-role-source>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</ejb-jar>
```

Все отношения между компонентом Customer и другими объектными компонентами, такими как CreditCard, Address и Phone, требуют, чтобы мы в дополнение к абстрактным методам доступа определили элемент <ejb-relation>.

У каждого отношения может быть имя, задаваемое элементом <ejb-relation-name>. Оно служит целям идентификации отношений для лиц, читающих дескриптор развертывания, или для инструментальных средств развертывания. Это имя не является обязательным.

Каждый элемент <ejb-relation> содержит ровно два элемента <ejb-relationship-role>, по одному на каждого участника отношений. В предыдущем примере первый элемент <ejb-relationship-role> описывал

роль компонента Customer в отношении. Она нам известна, потому что элемент <relationship-role-source> определяет CustomerEJB в элементе <ejb-name>. CustomerEJB – это имя компонента, используемое в первоначальном объявлении компонента Customer в разделе <enterprise-beans>. Значение элемента <ejb-name> в разделе <relationship-role-source> должно всегда совпадать со значением элемента <ejb-name> в разделе <enterprise-beans>.

Элемент <ejb-relationship-role> также объявляет количество элементов, или *множественность* (*multiplicity*), роли. Элемент <multiplicity> может иметь либо значение One, либо Many. В нашем случае элементу <multiplicity> компонента Customer присвоено значение One, показывающее, что каждый компонент Address имеет отношение ровно с *одним* компонентом Customer. Элемент <multiplicity> компонента Address тоже определен как One, а это означает, что каждый компонент Customer имеет отношение ровно с одним компонентом Address. Если бы у компонента Customer были отношения с многими компонентами Address, для элемента <multiplicity> компонента Address было бы установлено значение Many.

В главе 6 мы определили компонент Customer, содержащий абстрактные методы доступа для получения и установки компонента Address в поле homeAddress, но у компонента Address нет абстрактных методов доступов для компонента Customer. В данном случае компонент Customer поддерживает ссылку на компонент Address, а компонент Address не поддерживает обратную ссылку на компонент Customer. Это называется однонаправленным отношением, означающим, что в этом отношении только один из объектных компонентов поддерживает управляемое контейнером поле отношения.

Если компонент, описанный элементом <ejb-relationship-role>, поддерживает ссылку на другой компонент в отношении, эта ссылка должна быть объявлена в элементе <cmr-field> как *управляемое контейнером поле отношений* (*container-managed relationship field*). Элемент <cmr-field> объявлен внутри элемента <ejb-relationship-role>:

```
<ejb-relationship-role>
    <ejb-relationship-role-name>
        Customer-has-an-Address
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <ejb-name>CustomerEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <cmr-field-name>homeAddress</cmr-field-name>
   </cmr-field>
</ejb-relationship-role>
```

EJB 2.0 требует, чтобы значение элемента <cmr-field-name> начиналось со строчной буквы. Для каждого поля отношений, определенного элементом <cmr-field>, в классе компонента должна существовать соответствующая ему пара абстрактных методов доступа. Один метод этой пары должен иметь имя set<cmr-field-name>() с первым символом значения <cmr-field-name>, преобразованным к верхнему регистру. Другой метод определяется как get<cmr-field-name>(), также с первым символом значения поля <cmr-field-name> в верхнем регистре. В предыдущем примере поле <cmr-field-name> установлено в значение homeAddress, соответствующее абстрактным методам доступа getHomeAddress() и setHomeAddress(), определенным в классе CustomerBean:

```
// Код класса компонента
public abstract void setHomeAddress(AddressLocal address);
public abstract AddressLocal getHomeAddress();

// Объявление в XML-дескрипторе развертывания
<cmr-field>
    <cmr-field-name>homeAddress</cmr-field-name>
</cmr-field>
```

Тип возвращаемого значения метода get<cmr-field-name>() должен совпадать с типом параметра метода set<cmr-field-name>(). Этот тип должен быть локальным интерфейсом объектного компонента, ссылающимся на один из двух типов java.util.Collection. В случае с полем отношений homeAddress мы используем локальный интерфейс компонента Address – AddressLocal. Типы коллекций (collections) рассмотрены более подробно далее в этой главе в разделах, посвященных отношениям «один-ко-многим», «многие-к-одному» и «многие-ко-многим».

Итак, у нас есть базовое понимание механизма описания элементов в абстрактной схеме постоянства, и мы готовы рассмотреть каждый из семи типов отношений более подробно. В ходе этого процесса мы представим дополнительные объектные компоненты, такие как CreditCard, Phone, Ship и Reservation, поддерживающие отношения с компонентом Customer.

Важно понимать, что хотя у объектных компонентов могут быть и локальный, и удаленный интерфейсы, управляемое контейнером поле отношений при поддержании отношения может использовать только локальный интерфейс объектного компонента. Так, нельзя определить абстрактный метод доступа, который имел бы параметр с типом javax.ejb.EJBObject (тип удаленного интерфейса). Все управляемые контейнером отношения основаны на типах javax.ejb.EJBLocalObject (локальный интерфейс).

Моделирование базы данных

В этой главе рассматриваются несколько разных схем таблиц базы данных. Эти схемы предназначены только для демонстрации возмож-

ных отношений между объектами в базе данных и не являются рекомендуемыми. Например, отношение Address–Customer обозначено наличием в таблице CUSTOMER поддержки внешнего ключа таблицы ADDRESS. Это не похоже на организацию большинства баз данных, в которых, вероятно, используется таблица связей или вводится поддержка внешнего ключа таблицы CUSTOMER в таблицу ADDRESS. Однако эта схема показывает, как постоянство, управляемое контейнером, в EJB 2.0 может поддерживать разные структуры баз данных.

На протяжении всей этой главы мы будем исходить из допущения, что таблицы базы данных созданы раньше приложения EJB, – другими словами, что приложение EJB обращается к существующей базе данных. Некоторые производители предлагают инструментальные средства, автоматически генерирующие таблицы в соответствии с отношениями, определенными для объектных компонентов. Эти инструментальные средства могут создавать схемы, которые сильно отличаются от рассматриваемых здесь. В других случаях производители, поддерживающие автоматическое создание схем базы данных, могут не проявлять гибкость, достаточную для поддержки схем, показанных в этой главе. Как разработчик EJB вы, в свою очередь, должны обладать достаточной приспосабливаемостью, для того чтобы адаптироваться к средствам, предоставленным поставщиком EJB.

Однонаправленное отношение «один-к-одному»

Примером однонаправленных отношений «один-к-одному» являются отношения между компонентами Customer (Пассажир) и Address (Адрес), определенным в главе 6. В этом случае у каждого компонента Customer есть только один Address, а у каждого компонента Address есть только один Customer. Который из этих двух компонентов ссылается на другой, определяется направлением движения. Если Customer владеет ссылкой на Address, Address не может ссылаться на Customer. Такие отношения называются однонаправленными, т. к. двигаться можно только по направлению от компонента Customer к компоненту Address, а не наоборот. Другими словами, компонент Address не имеет никакого представления, кому он принадлежит. Этот тип отношений показан на рис. 7.1.

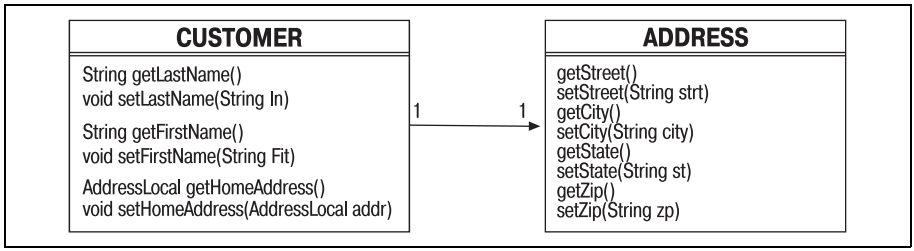

Рис. 7.1. Однонаправленное отношение «один-к-одному»

Схема реляционной базы данных

Как показано на рис. 7.2, однонаправленные отношения «один-к-одному» обычно реализуют достаточно типичную схему реляционной базы данных, в которой таблица содержит внешний ключ (указатель) на другую таблицу. В этом случае таблица CUSTOMER содержит внешний ключ на таблицу ADDRESS, а таблица ADDRESS не содержит внешний ключ на таблицу CUSTOMER. Это позволяет записям таблицы ADDRESS совместно использоваться другими таблицами. Такой сценарий рассматривается в разделе, посвященном однонаправленным отношениям «многие-ко-многим». Тот факт, что схема базы данных – не то же самое, что абстрактная схема постоянства, иллюстрирует некоторую их независимость.

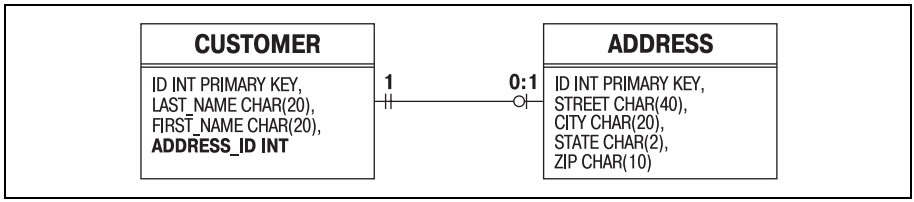

Рис. 7.2. Однонаправленное отношение в СУРБД

Абстрактная программная модель

Как мы узнали из главы 6, для определения полей отношений в классе компонента применяются абстрактные методы доступа. Если объектный компонент поддерживает ссылку на другой компонент, то для моделирования этой ссылки он определяет пару абстрактных методов доступа. В однонаправленных отношениях эти абстрактные методы доступа определяются только в одном из компонентов. Таким образом, для обращения к компоненту Address можно вызывать методы getHomeAddress()/setHomeAddress() из класса CustomerBean, но не существует никаких методов для доступа к компоненту Customer из класса AddressBean.

Компонент Address может совместно использоваться несколькими полями отношений одного компонента, но он не может совместно использоваться несколькими компонентами Customer. Если, например, компонент Customer определяет два поля отношений, billingAddress и homeAddress (однонаправленные отношения с компонентом Address), эти два поля могут ссылаться на один компонент Address:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
   public void setAddress(String street,String city,String state,String zip)
{
        ...
        address = addressHome.createAddress(street, city, state, zip);
        this.setHomeAddress(address);
```

```
            this.setBillingAddress(address);

            AddressLocal billAddr = this.getBillingAddress();
            AddressLocal homeAddr = this.getHomeAddress();

            if(billAddr.isIdentical(homeAddr))
           // всегда true

            ...
        }
        ...
    }
```

Если когда-нибудь нам понадобится сделать `billingAddress` отличным от `homeAddress`, мы сможем просто связать его с другим компонентом Address. Однако иногда совместное использование ссылки на другой компонент двумя полями отношений в одном и том же объекте очень удобно. Для поддержки этого типа отношений к таблице `CUSTOMER` может быть добавлено новое поле адреса для счетов (billing address):

```
CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
    ADDRESS_ID INT,
    BILLING_ADDRESS_ID INT
}
```

Как было показано в предыдущем примере, возможно, чтобы два поля компонента (в этом случае – поля `homeAddress` и `billingAddress` компонента Customer) ссылались на одно и то же отношение (т. е. на один компонент Address) в случае, если тип этих отношений одинаковый. Однако совместное использование одного компонента Address двумя разными компонентами Customer невозможно. Если бы, например, домашний адрес компонента Customer A был связан с домашним адресом Customer B, а затем этот адрес был бы изменен, то у компонента Customer A не стало бы домашнего адреса. Как показано на рис. 7.3, компонент Address 2 первоначально связан с компонентом Customer B, но отсоединяется при переназначении компонента Address 1 компоненту Customer B.

Этот на первый взгляд странный побочный эффект является просто естественным результатом определения отношений. Отношение компонентов Customer–Address было определено как «один-к-одному», поэтому на компонент Address может ссылаться только один компонент Customer.

Если с полем `AddressHome` компонента Customer не связан компонент Address, то метод `getHomeAddress()` возвратит пустой указатель. Это справедливо для всех полей отношений, управляемых контейнером, ссылающихся на один объектный компонент.

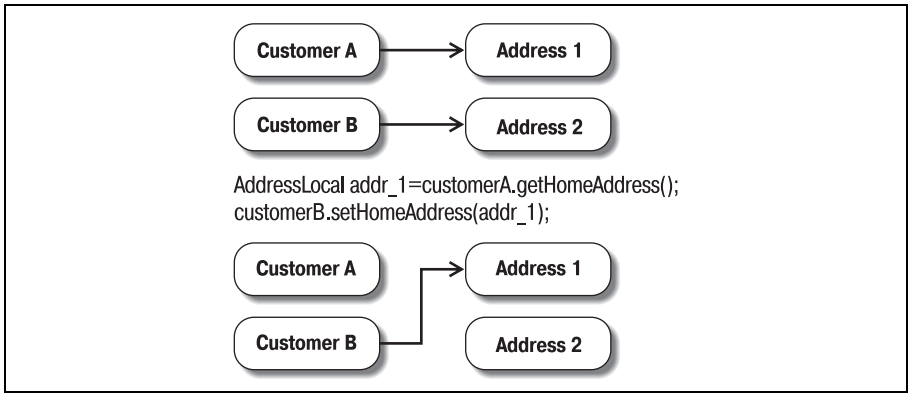

Рис. 7.3. Обмен ссылками в однонаправленных отношениях «один-к-одному»

Абстрактная схема постоянства

Ранее в этой главе мы уже определили элементы XML для отношения Клиент–Адрес, поэтому здесь мы не будем их повторять. Элемент `<ejb-relation>` из соответствующего раздела описывает однонаправленные отношения «один-к-одному». Однако если компонент Customer поддерживает два поля отношений с компонентом Address – `homeAddress` и `billingAddress`, то каждое из этих отношений должно быть описано в своем собственном элементе `<ejb-relation>`:

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-HomeAddress</ejb-relation-name>
        <ejb-relationship-role>
            ...
            <cmr-field>
                <cmr-field-name>homeAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            ...
        </ejb-relationship-role>
    </ejb-relation>
    <ejb-relation>
        <ejb-relation-name>Customer-BillingAddress</ejb-relation-name>
        <ejb-relationship-role>
            ...
            <cmr-field>
                <cmr-field-name>billingAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            ...
        </ejb-relationship-role>
```

```
        </ejb-relation>
    </relationships>
```

Двунаправленное отношение «один-к-одному»

Мы можем расширить наш компонент Customer и включить в него ссылку на компонент CreditCard, содержащую информацию о кредитной карточке. Компонент Customer будет содержать ссылку на свой компонент CreditCard, а компонент CreditCard будет поддерживать обратную ссылку на компонент Customer. Это имеет смысл, т. к. CreditCard должен знать, какому компоненту он принадлежит. А поскольку у каждого компонента CreditCard есть обратная ссылка на один компонент Customer, и каждый Customer ссылается на один CreditCard, то мы имеем двунаправленные отношения «один-к-одному».

Схема реляционной базы данных

Для компонента CreditCard создана соответствующая таблица CREDIT_CARD, поэтому мы должны добавить к CREDIT_CARD внешний ключ на таблицу CUSTOMER:

```
CREATE TABLE CREDIT_CARD
(
    ID INT PRIMARY KEY NOT NULL,
    EXP_DATE DATE,
    NUMBER CHAR(20),
    NAME CHAR(40),
    ORGANIZATION CHAR(20),
    CUSTOMER_ID INT
}

CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
    HOME_ADDRESS_ID INT,
    ADDRESS_ID INT,
    CREDIT_CARD_ID INT
)
```

Двунаправленные отношения «один-к-одному» могут моделировать схемы реляционной базы данных, в которых две таблицы содержат внешние ключи друг на друга (в частности, две строки из этих таблиц указывают друг на друга). На рис. 7.4. показано, как эта схема должна быть реализована для строк в таблицах CUSTOMER и CREDIT_CARD.

Двунаправленные отношения «один-к-одному» могут быть установлены также с помощью таблицы связей (linking table), в которой каждый столбец внешнего ключа должен быть уникален. Это удобно, если мы не хотим реализовывать отношения внутри первоначальных таб-

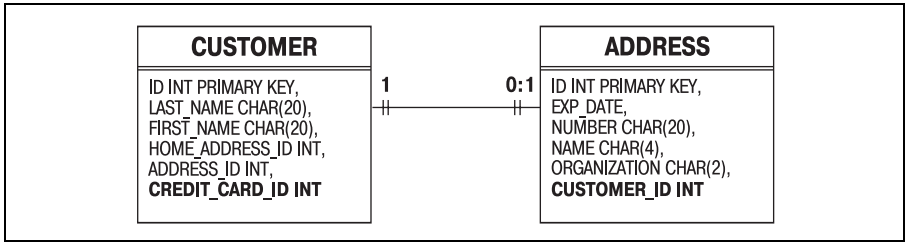

Рис. 7.4. Двунаправленные отношения «один-к-одному» в СУРБД

лиц. Таблицы связей будут использованы позже в этой главе для установки отношений «один-ко-многим» и «многие-ко-многим», но сейчас важно запомнить, что схема базы данных в этих примерах имеет чисто иллюстративный характер. Абстрактная схема постоянства объектного компонента может соответствовать нескольким схемам базы данных, а схема базы данных в этих примерах представляет собой только одну из возможных.

Абстрактная программная модель

Для моделирования отношения между компонентами Customer и CreditCard мы должны объявить в классе `CreditCardBean` поле отношения с именем `customer`:

```
public abstract class CreditCardBean extends javax.ejb.EntityBean {

    ...

    // поля отношений
    public abstract CustomerLocal getCustomer();
    public abstract void setCustomer(CustomerLocal local);

    // поля постоянства
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract Date getExpirationDate();
    public abstract void setExpirationDate(Date date);
    public abstract String getNumber();
    public abstract void setNumber(String number);
    public abstract String getNameOnCard();
    public abstract void setNameOnCard(String name);
    public abstract String getCreditOrganization();
    public abstract void setCreditOrganization(String org);

    // стандартные методы обратного вызова
    ...

}
```

В данном случае мы использовали локальный интерфейс компонента Customer (предполагаем, что он был создан) из-за того, что поля отно-

шений требуют применения типов локальных интерфейсов. Все отношения, рассматриваемые в оставшейся части этой главы, подразумевают локальные интерфейсы. Конечно, ограничившись локальными интерфейсами вместо удаленных, мы лишаемся прозрачности расположения. Все объектные компоненты должны быть расположены в одном и том же процессе или виртуальной машине Java (JVM). Хотя поля отношений, использующие удаленные интерфейсы, не поддерживаются в EJB 2.0, вероятно, что поддержка удаленных полей отношений будет добавлена в последующие версии спецификации.

Мы также можем добавить набор абстрактных методов доступа к полю отношения `creditCard` в класс `CustomerBean`:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
    public abstract void setCreditCard(CreditCardLocal card);
    public abstract CreditCardLocal getCreditCard();
    ...
}
```

Хотя метод `setCustomer()` и доступен в классе `CreditCardBean`, мы не должны явно устанавливать ссылку на Customer в компоненте CreditCard. Когда ссылка на компонент CreditCard передается в метод `setCreditCard()` класса `CustomerBean`, контейнер EJB автоматически устанавливает отношение с компонентом Customer в компоненте CreditCard для обеспечения обратной ссылки на данный компонент:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
    public void setCreditCard(Date exp, String numb, String name, String org)
        throws CreateException {
        ...

        card = creditCardHome.create(exp,numb,name,org);

        // поле customer компонента CreditCard будет установлено автоматически
        this.setCreditCard(card);

        Customer customer = card.getCustomer();

        if(customer.isIdentical(ejbContext.getEJBLocalObject())
        // всегда true

        ...
    }
    ...
}
```

Аналогично правилам для однонаправленных отношений «один-к-одному», компонент CreditCard в двунаправленных отношениях «один-к-одному» может совместно использоваться несколькими полями отношений одного компонента Customer, и не может – разными компо-

нентами Customer. Как показано на рис. 7.5, назначение компонента `CreditCard` клиента Customer A клиенту Customer B разрывает связь компонента `CreditCard` с Customer A и связывает его с Customer B.

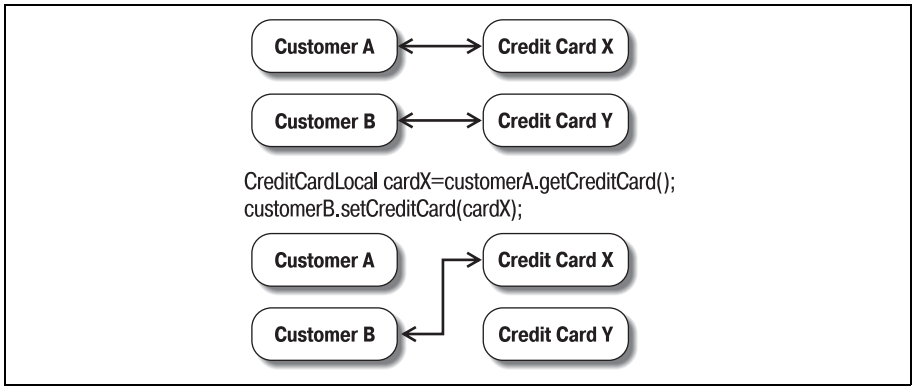

Рис. 7.5. Обмен ссылками в двунаправленных отношениях «один-к-одному»

Абстрактная схема постоянства

Элемент `<ejb-relationship>`, описывающий отношение Customer–CreditCard, походит на тот, которым мы оперировали для отношения Customer–Address, но с одним важным различием – оба элемента `<ejb-relationship-role>` содержат элемент `<cmr-field>`:

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-CreditCard</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-a-CreditCard
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>creditCard</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                CreditCard-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CreditCardEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
```

```
                <cmr-field-name>customer</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

Тот факт, что оба участника отношений определяют элементы <cmr-field> (поля отношений), сообщает нам, что отношение является двунаправленным.

Однонаправленное отношение «один-ко-многим»

Объектные компоненты также могут поддерживать множественные отношения. Это означает, что один объектный компонент может объединять (или содержать) несколько других объектных компонентов. Например, компонент Customer может установить отношения с несколькими компонентами Phone, каждый из которых представляет номер телефона. Это сильно отличается от простых отношений «один-к-одному». Отношения «один-ко-многим» и «многие-ко-многим» требуют, чтобы при доступе к полю отношения разработчик оперировал коллекциями ссылок, а не отдельной ссылкой.

Схема реляционной базы данных

Для иллюстрации однонаправленного отношения «один-ко-многим» мы будем использовать новый объектный компонент – Phone, для которого нам необходимо определить таблицу Phone:

```
CREATE TABLE PHONE
(
    ID INT PRIMARY KEY NOT NULL,
    NUMBER CHAR(20),
    TYPE INT,
    CUSTOMER_ID INT
}
```

Однонаправленные отношения «один-ко-многим» между таблицами CUSTOMER и PHONE в реляционной базе данных могут быть выражены множеством способов. Для этого примера мы решили включить в таблицу PHONE внешний ключ, ссылающийся на таблицу CUSTOMER.

Таблица связываемых данных может поддерживать столбец неуникальных внешних ключей на таблицу связей. В случае с компонентами Customer и Phone таблица PHONE поддерживает внешний ключ на таблицу CUSTOMER, а одна или несколько записей PHONE могут содержать внешние ключи на ту же запись в CUSTOMER. Другими словами, в этой базе данных записи в PHONE указывают на записи в CUSTOMER. Однако, согласно абстрактной программной модели, компонент Customer должен указывать на компоненты Phone – т. е. эти две схемы полностью обратны. Как же это работает? Контейнерная система скрывает обрат-

ный указатель так, чтобы казалось, будто Customer знает о компоненте Phone, а не наоборот. Когда мы просим контейнер вернуть коллекцию (Collection) компонентов Phone (вызывая метод `getPhoneNumbers()`), он запрашивает из таблицы `PHONE` все записи с внешним ключом, совпадающим с первичным ключом компонента Customer. Применение обратных указателей в этом типе отношения проиллюстрировано на рис. 7.6.

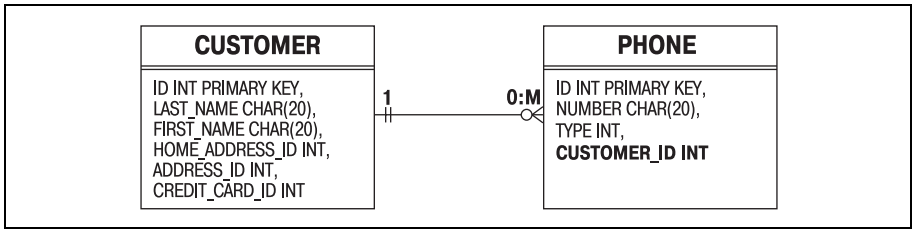

Рис. 7.6. Однонаправленное отношение «один-ко-многим» в СУРБД с использованием обратных указателей

Эта схема базы данных показывает, что структура и отношения в реальной базе данных могут отличаться от отношений, определенных в абстрактной программной модели. В данном случае таблицы оказались перевернуты, но контейнерная система EJB будет управлять компонентами в соответствии с указаниями разработчика компонента. Это не всегда возможно; иногда схема базы данных несовместима с требуемым полем отношения. Однако когда мы имеем дело с существующими базами данных (т. е. с базами данных, которые были установлены до приложения EJB), сценарий с обратным указателем, похожий на показанный здесь, достаточно распространен, поэтому поддержка такого вида отношений очень важна.

Более простая реализация отношения Customer–Phone могла бы использовать таблицу связей, поддерживающую два столбца внешних ключей, указывающих на записи и в таблице `PHONE`, и в `CUSTOMER`. Мы могли бы тогда наложить на столбец внешнего ключа `PHONE` в таблице связи уникальное ограничение, чтобы гарантировать, что он будет содержать только уникальные значения (иначе говоря, что каждый телефон имеет только одного клиента), в то же время разрешая столбцу внешнего ключа `CUSTOMER` содержать дубликаты. Преимущество таблицы связей состоит в том, что она не налагает отношение между записями `CUSTOMER` и `PHONE` ни на одну из этих таблиц.

Абстрактная программная модель

В абстрактной программной модели мы описываем множественность, определяя поле отношения, которое может указывать на несколько объектных компонентов. Для того чтобы сделать это, мы обратимся к тем же абстрактным методам доступа, которые мы применяли и для отношений «один-к-одному», но на этот раз установим тип поля либо

в java.util.Collection, либо в java.util.Set. Объект Collection содержит однородную группу ссылок на локальный компонентный объект, а это значит, что он содержит несколько ссылок на объектные компоненты одного типа. Тип Collection может содержать дублирующиеся ссылки на один и тот же объектный компонент, в то время как тип Set не может.

Так, компонент Customer может поддерживать отношения с несколькими номерами телефонов (например, с домашним, рабочим, сотовым, факсом и т. д.), представленными компонентом Phone. Компонент Customer не содержит отдельные поля отношения для каждого из этих компонентов Phone, а сохраняет все компоненты Phone в поле отношения, основанном на коллекции, доступном посредством абстрактных методов доступа:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...
    // Поля отношений
    public java.util.Collection getPhoneNumbers();
    public void setPhoneNumbers(java.util.Collection phones);

    public AddressLocal getHomeAddress();()
    public void setHomeAddress(AddressLocal local);
    ...
```

Как и у всех объектных компонентов, у компонента Phone есть класс компонента и локальный интерфейс, как показано в следующем листинге. Обратите внимание, что класс PhoneBean не содержит поле отношения для компонента Customer. Это отношение – однонаправленное; Customer поддерживает отношения с несколькими компонентами Phone, а компоненты Phone не поддерживают обратного поля отношения для Customer. О существовании этого отношения знает только компонент Customer:

```
// Локальный интерфейс компонента Phone
public interface PhoneLocal extends javax.ejb.EJBLocalObject {
    public String getNumber();
    public void setNumber(String number);
    public byte getType();
    public void setType(byte type);
}

// Класс компонента Phone
public class PhoneBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(String number, byte type) {
        setNumber(number);
        setType(type);
        return null;
    }
    public void ejbPostCreate(String number,byte type) {
    }
```

```
    // Поля постоянства
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract String getNumber();
    public abstract void setNumber(String number);
    public abstract byte getType();
    public abstract void setType(byte type);

    // Стандартные методы обратного вызова
    ...
}
```

Для того чтобы показать, как объектный компонент использует поле отношения, основанное на коллекции, мы создадим в классе CustomerBean метод, позволяющий удаленным клиентам добавлять новые номера телефонов. Этот метод, addPhoneNumber(), использует параметры номера телефона для того, чтобы создать новый компонент Phone, а затем добавить его к полю отношения, основанному на коллекции, с именем phoneNumbers:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // Прикладные методы
    public void addPhoneNumber(String number, String type) {

        InitialContext jndiEnc = new InitialContext();
        PhoneHomeLocal phoneHome = jndiEnc.lookup("PhoneHomeLocal");
        PhoneLocal phone = phoneHome.create(number,type);

        Collection phoneNumbers = this.getPhoneNumbers();
        phoneNumbers.add(phone);

    }
    ...
    // Поля отношений
    public java.util.Collection getPhoneNumbers();
    public void setPhoneNumbers(java.util.Collection phones);

    ...
```

Обратите внимание, что мы сначала создали компонент Phone, а затем добавили его к основанному на коллекции отношению phoneNumbers. Мы получили объект phoneNumbers типа Collection из метода доступа getPhoneNumbers(), а затем добавили к нему новый компонент Phone точно так же, как мы добавляем к коллекции любой другой объект. Добавление компонента Phone к коллекции заставляет контейнер EJB установить в новой записи PHONE внешний ключ так, чтобы он указывал обратно на запись таблицы CUSTOMER, относящуюся к компоненту Customer. Если бы мы использовали таблицу связи, была бы создана новая связующая запись. Начиная с этого момента, новый компонент Phone будет доступен для основанного на коллекции отношения phoneNumbers.

Можно также посредством метода доступа модифицировать или удалять ссылки из основанного на коллекции поля отношений. Напри-

мер, следующий код определяет в классе CustomerBean два метода, позволяющие клиентам удалять или модифицировать номера телефонов в поле отношения phoneNumbers компонента:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // Прикладные методы
    public void removePhoneNumber(byte typeToRemove) {

        Collection phoneNumbers = this.getPhoneNumbers();
        Iterator iterator = phoneNumbers.iterator();
        while(iterator.hasNext()) {
            PhoneLocal phone = (PhoneLocal)iterator.next();
            if(phone.getType() = typeToRemove) {
                iterator.remove(phone);
                break;
            }
        }
    }
    public void updatePhoneNumber(String number,byte typeToUpdate) {
        Collection phoneNumbers = this.getPhoneNumbers();
        Iterator iterator = phoneNumbers.iterator();
        while(iterator.hasNext()) {
            PhoneLocal phone = (PhoneLocal)iterator.next();
            if(phone.getType() = typeToUpdate) {
                phone.setNumber(number);
                break;
            }
        }
    }
    ...
    // Поля отношений
    public java.util.Collection getPhoneNumbers();
    public void setPhoneNumbers(java.util.Collection phones);
```

В прикладном методе removePhoneNumber() компонент Phone с соответствующим типом был найден, а затем удален из основанного на коллекции отношения. Номер телефона не был удален из базы данных, он просто был отсоединен от компонента Customer (т. е. он больше не ссылается на Customer). На рис. 7.7 показано, что происходит, если из отношения, основанного на коллекции, удаляется ссылка на компонент Phone.

Метод updatePhoneNumber() действительно изменяет существующий компонент Phone, меняя его состояние в базе данных. Отношение, основанное на коллекции, все еще ссылается на компонент Phone, но его данные уже были изменены.

Методы RemovePhoneNumber() и updatePhoneNumber() иллюстрируют тот факт, что к отношениям на базе коллекции можно обращаться и модифицировать их точно так же, как и любой другой объект Collection. Кроме того, из объекта Collection для выполнения циклических опе-

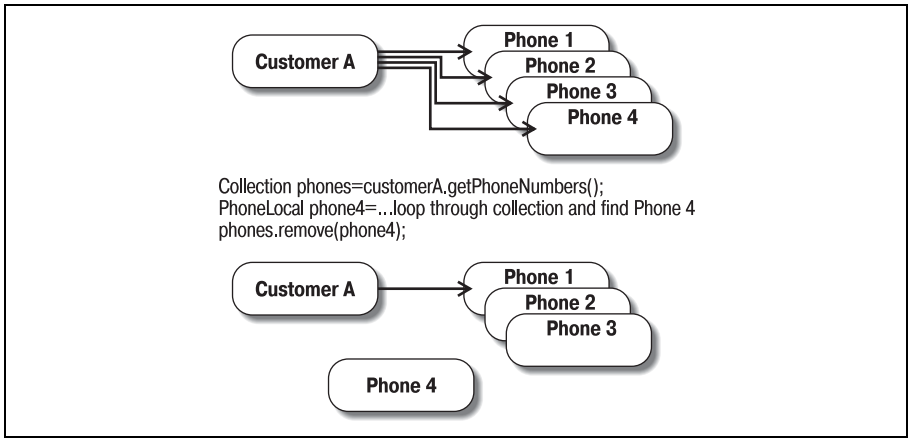

Рис. 7.7. Удаление ссылки на компонент из поля отношения типа «коллекция»

раций может быть получен объект `java.util.Iterator`. Однако в случае применения итератора с основанным на коллекции полем отношений можно получить предупреждение. Нельзя во время работы с объектом `Iterator` добавлять или удалять элементы из объекта `Collection`. Единственное исключение из этого правила состоит в том, что для удаления элемента может быть вызван метод `Iterator.remove()`. Хотя методы `Collection.add()` и `Collection.remove()` и могут применяться в других обстоятельствах, вызов этих методов во время работы с итератором приведет к возбуждению исключения `java.util.IllegalStateException`.

Если к полю отношения `phoneNumbers` не были добавлены никакие компоненты, метод `getPhoneNumbers()` возвратит пустой объект `Collection`. Поля отношения `<multiplicity>` никогда не возвращают пустой указатель. Объект `Collection`, используемый полем отношения, реализуется контейнерной системой, специфичной для конкретного производителя, и сильно связан с внутренними механизмами контейнера. Это позволяет контейнеру EJB принимать меры по улучшению производительности, такие как отложенная загрузка и оптимистический параллелизм, скрывая от разработчика компонентов эти нестандартные механизмы.[1] Определенные приложением объекты `Collection` могут использоваться с полями отношений, управляемых контейнером, только если элементы имеют правильный тип. Например, можно создать новый объект `Collection` и затем с помощью метода `setPhoneNumbers()` добавить его к компоненту Customer:

```
public void addPhoneNumber(String number, String type) {
    ...
```

[1] Объект Collection из отношения на базе коллекции при его использовании в транзакции не может быть модифицирован за пределами зоны этой транзакции. Подробности см. в главе 14.

```
        PhoneLocal phone = phoneHome.create(number,type);

        Collection phoneNumbers = java.util.Vector();
        phoneNumbers.add(phone);

        // Это допускается
        this.setPhoneNumbers(phoneNumbers);

    }

    // Поля отношений
    public java.util.Collection getPhoneNumbers();

    public void setPhoneNumbers(java.util.Collection phones);
```

Мы часто использовали метод `getPhoneNumbers()`, но еще не обращались к методу `setPhoneNumbers()`. В большинстве случаев этот метод не будет применяться, т. к. он обновляет всю коллекцию номеров телефонов. Однако он может быть полезен для обмена отношениями между объектными компонентами.

Если два компонента Customer хотят обменяться номерами телефонов, они могут сделать это множеством разных способов. Самый важный момент, который следует иметь в виду, состоит в том, что компонент Phone как субъект однонаправленного отношения «один-ко-многим» может ссылаться только на один компонент Customer. Он может быть скопирован, и, таким образом, оба компонента Customer будут иметь компоненты Phone с одинаковыми данными, но сам компонент Phone не может использоваться ими совместно.

Представьте, например, что компонент Customer A хочет передать все свои телефонные номера компоненту Customer B. Он может сделать это с помощью метода `setPhoneNumbers()` компонента Customer B, как показано в следующем листинге (мы предполагаем, что компоненты Customer взаимодействуют через свои локальные интерфейсы):

```
CustomerLocal customerA = ... get Customer A
CustomerLocal customerB = ... get Customer B

Collection phonesA = customerA.getPhoneNumbers();
customerB.setPhoneNumbers( phonesA );

if( customerA.getPhoneNumbers().isEmpty())
    // это будет true
if( phonesA.isEmpty()) )
    // это будет true
```

Как показано на рис. 7.8, передача одного отношения, основанного на коллекции, другому приводит фактически к отсоединению этих отношений от первого компонента и привязывает их ко второму. Кроме того, если второй компонент уже содержит в своем поле отношения `phoneNumbers` коллекцию компонентов Phone, эти компоненты выкидываются из поля отношения и отсоединяются от компонента.

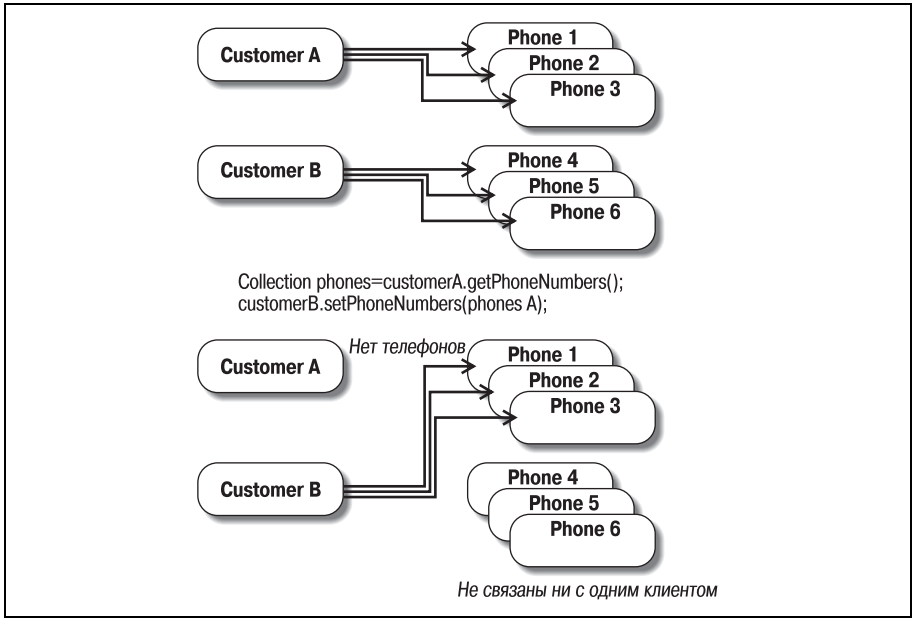

Рис. 7.8. Обмен коллекциями отношений в однонаправленном отношении «один-ко-многим»

Результат этого обмена может быть не совсем понятен, но необходимо поддержать множественность отношения, которая гласит, что у компонента Phone может быть только один компонент Customer. Это объясняет, почему компоненты Phone 1, 2 и 3 не ссылаются на оба компонента Customer A и B, но не объясняет, почему компоненты Phone 4, 5, и 6 отсоединяются от Customer B. Почему Customer B не связан со всеми компонентами Phone? Причина – исключительно вопрос семантики, т. к. схема реляционной базы данных технически не смогла бы предотвратить это событие. Факт замены одного объекта `Collection` другим с помощью вызова `setPhoneNumbers(Collection collection)` подразумевает, что на начальный объект `Collection` компонента Customer B больше нет ссылок.

Кроме полного перемещения отношений на базе коллекции между компонентами, также можно перемещать отдельные компоненты Phone между компонентами Customer. Они тоже не могут использоваться совместно. Например, если компонент Phone, связанный с Customer, добавлен к коллекции отношений Customer B, этот компонент Phone будет изменен таким образом, что будет ссылаться вместо Customer A на Customer B, как показано на рис. 7.9.

И еще раз, множественность отношений не дает компоненту Phone 1 ссылаться одновременно и на компонент Customer A, и на Customer B.

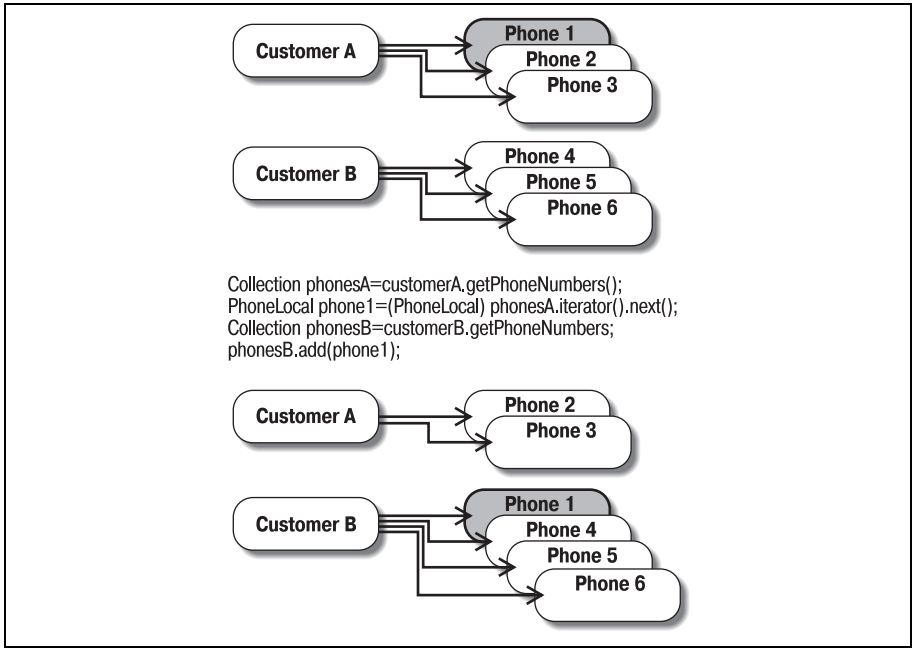

Рис. 7.9. Обмен компонентом в однонаправленном отношении «один-ко-многим»

Абстрактная схема постоянства

Абстрактная схема постоянства для однонаправленных отношений «один-ко-многим» имеет несколько существенных отличий в элементах `<ejb-relation>`, как показано ниже:

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-Phones</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-many-Phone-numbers
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>phoneNumbers</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Phone-belongs-to-Customer
```

```
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>PhoneEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

В элементе <ejb-relation> множественность компонента Customer объявлена как One, тогда как множественность элемента <ejb-relationship-role> компонента Phone содержит Many. Это, конечно же, устанавливает отношение «один-ко-многим». Тот факт, что <ejb-relationship-role> компонента Phone не включает элемент <cmr-field>, указывает на то, что отношение «один-ко-многим» является однонаправленным; компонент Phone не содержит обратную ссылку на компонент Customer.

Наиболее интересное изменение состоит в том, что в объявление <cmr-field> компонента Customer добавляется элемент <cmr-field-type>, который должен быть определен для компонента, имеющего основанное на коллекции поле отношения (в данном случае – поле phoneNumbers, поддерживаемое компонентом Customer), и может принимать одно из двух значений – java.util.Collection или java.util.Set, являющихся разрешенными типами для основанных на коллекциях отношений. В будущей спецификации разрешенные типы могут быть расширены с целью включения java.util.List и java.util.Map, но пока они еще не поддерживаются.

📖 Рабочее упражнение 7.1. Отношения объектов в CMP 2.0: Часть 1

Компоненты Cruise, Ship и Reservation

Для того чтобы сделать изучение более интересным, нам нужно ввести еще несколько объектных компонентов, что позволит нам промоделировать оставшиеся четыре отношения: однонаправленные «многие-к-одному», двунаправленные «один-ко-многим», двунаправленные «многие-ко-многим» и однонаправленные «многие-ко-многим».

В системе заказа билетов «Титан» каждый пассажир (Customer) может быть внесен в список одного или нескольких круизов (Cruises). Каждая регистрация требует заказа билетов (Reservation). Заказ может быть сделан для одного или нескольких (обычно двух) пассажиров. Для каждого круиза требуется ровно одно судно (Ship), но каждое судно может в течение года участвовать в нескольких круизах. Эти отношения показаны на рис. 7.10.

Каждое отношение, исследуемое в следующих четырех разделах, будет ссылаться на вышеприведенную диаграмму.

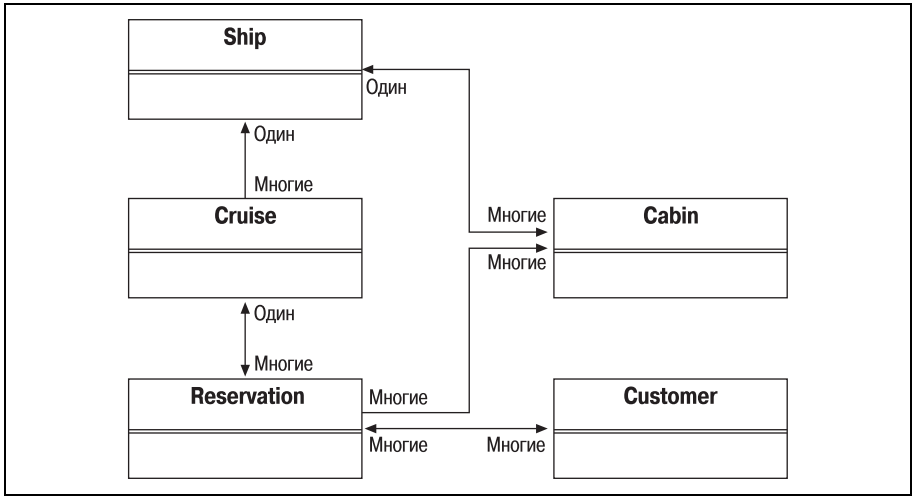

Рис. 7.10. Диаграмма классов Cruise, Ship, Reservation, Cabin и Customer

Однонаправленное отношение «многие-к-одному»

Однонаправленные отношения «многие-к-одному» имеют место, когда несколько объектных компонентов ссылаются на один объектный компонент, но этот компонент не знает об этом отношении. В фирме «Titan Cruise», например, понятие круиза может быть реализовано компонентом Cruise. Как показано на рис. 7.10, каждый компонент Cruise поддерживает отношение типа «многие-к-одному» с компонентом Ship. Это отношение является однонаправленным; Компонент Cruise поддерживает отношение с компонентом Ship, но компонент Ship не отслеживает компоненты Cruise, которыми он используется.

Схема реляционной базы данных

Схема реляционной базы данных для отношения Cruise-to-Ship достаточно проста; она требует, чтобы таблица `CRUISE` содержала столбец внешнего ключа на таблицу `SHIP` так, чтобы каждая строка таблицы `CRUISE` указывала на строку таблицы `SHIP`. Таблицы `CRUISE` и `SHIP` определены ниже. На рис. 7.11 показано отношение между этими таблицами в базе данных.

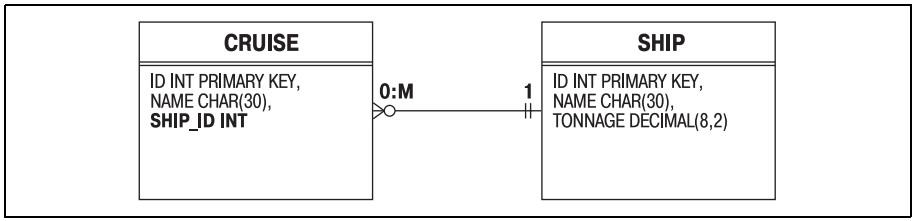

Рис. 7.11. Однонаправленное отношение «многие-к-одному» в СУРБД

Для адекватного описания океанского лайнера требовалось бы огромное количество данных, но здесь мы ограничимся простым определением таблицы SHIP:

```
CREATE TABLE SHIP
(
    ID INT PRIMARY KEY NOT NULL,
    NAME CHAR(30),
    TONNAGE DECIMAL (8,2)
}
```

Таблица CRUISE содержит данные о названии каждого круиза, судне и другую информацию, здесь не рассматриваемую. (Другие таблицы, такие как RESERVATION, SCHEDULE и CREW, могли бы иметь отношения с таблицей CRUISE через таблицу связей.) Не будем усложнять ее и сосредоточимся на определении, полезном для примеров в этой книге:

```
CREATE TABLE CRUISE
(
    ID INT PRIMARY KEY NOT NULL,
    NAME CHAR(30),
    SHIP_ID INT
}
```

Абстрактная программная модель

В этой абстрактной программной модели поле отношения имеет тип ShipLocal и поддерживается компонентом Cruise. Абстрактные методы доступа похожи на те, которые мы определяли в предыдущих примерах:

```
public abstract class CruiseBean implements javax.ejb.EntityBean {
    public Integer ejbCreate(String name, ShipLocal ship) {
        setName(name);
        return null;
    }
    public void ejbPostCreate(String name, ShipLocal ship) {
        setShip(ship);
    }
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract void setName(String name);
    public abstract String getName();
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip();

    // Методы обратного вызова
    ...
}
```

Обратите внимание, что компоненту Cruise требуется, чтобы при его создании в качестве параметра была передана ссылка ShipLocal; это

совершенно естественно, поскольку круиз не может существовать без судна. В соответствии со спецификацией EJB 2.0, поля отношения не могут изменяться или устанавливаться в методе `ejbCreate()`. Они должны модифицироваться методом `ejbPostCreate()`, – требование, которому следует класс `CruiseBean`.

Причина, по которой отношения устанавливаются в методе `ejbPostCreate()`, а не в `ejbCreate()`, проста: первичный ключ для объектного компонента не может быть доступен до выполнения метода `ejbCreate()`. Первичный ключ требуется в случае, если отображение для этого отношения использует данный ключ в качестве внешнего ключа, поэтому назначение отношений отложено до завершения метода `ejbPostCreate()`, после которого первичный ключ становится доступным. Это также применимо и к автогенерируемым первичным ключам, которые обычно требуют, чтобы вставка была сделана до того, как будет сгенерирован первичный ключ. Кроме этого, ссылочная целостность (referential integrity) может обязать использовать в зависимых таблицах ненулевые внешние ключи, поэтому вставка должна быть выполнена раньше всего. В действительности транзакция не завершается, пока не завершатся и метод `ejbCreate()`, и `ejbPostCreate()`, поэтому производители могут сами выбирать время для вставок в базу данных и связывания отношений.

Отношение между компонентами Cruise и Ship является однонаправленным, поэтому в компоненте Ship не определены никакие поля отношения, а только поля постоянства:

```
public abstract class ShipBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer primaryKey,String name,double tonnage) {
        setId(primaryKey);
        setName(name);
        setTonnage(tonnage);
        return null;
    }
    public void ejbPostCreate(Integer primaryKey,String name,double tonnage)
{
    }
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public abstract void setName(String name);
    public abstract String getName();
    public abstract void setTonnage(double tonnage);
    public abstract double getTonnage();

    // Методы обратного вызова
    ...
}
```

Теперь вам все должно быть уже понятно. Последствия обмена ссылками Ship между компонентами Cruise также должны быть очевидны.

Как показано ранее на рис. 7.10, каждый компонент Cruise может ссылаться только на один компонент Ship, но каждый Ship может ссылаться на несколько компонентов Cruise. Если мы возьмем Ship A, на который ссылается Cruise 1, и передадим его Cruise 4, то и Cruise 1, и Cruise 4 будут ссылаться на Ship A (рис. 7.12).

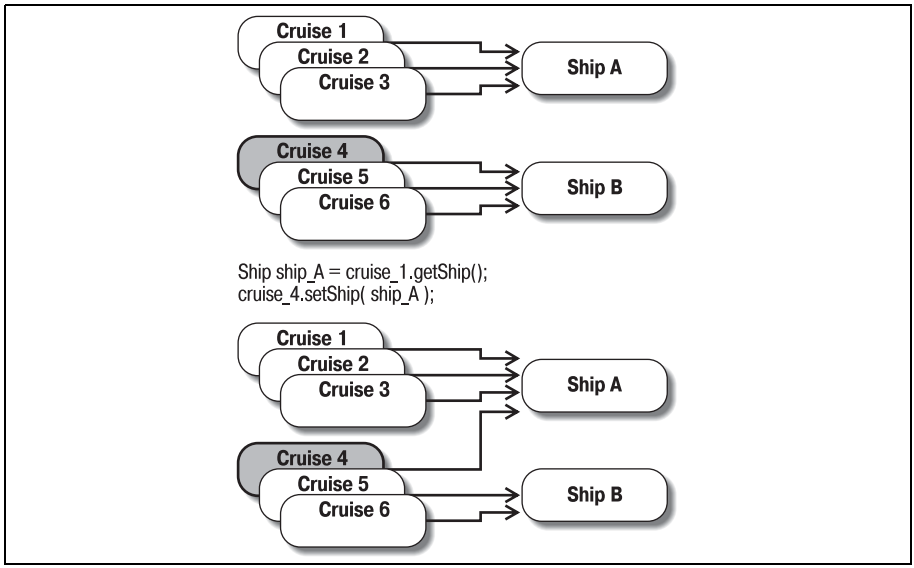

Рис. 7.12. Совместное использование ссылки на компонент в однонаправленном отношении «многие-к-одному»

Абстрактная схема постоянства

Абстрактная схема постоянства однонаправленного отношения «многие-к-одному» достаточно проста. Она основывается только на том, что вы уже знаете, и поэтому не содержит ничего неожиданного:

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CruiseEJB</ejb-name>
        <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
        <local>com.titan.cruise.CruiseLocal</local>
        ...
    </entity>
    <entity>
        <ejb-name>ShipEJB</ejb-name>
        <local-home>com.titan.ship.ShipHomeLocal</local-home>
        <local>com.titan.ship.ShipLocal</local>
        ...
    </entity>
    ...
</enterprise-beans>
```

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Cruise-Ship</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cruise-has-a-Ship
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>CruiseEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>ship</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Ship-has-many-Cruises
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>ShipEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

Элемент `<ejb-relationship-role>` компонента Cruise определяет его множественность как `Many`, а в качестве его поля отношения объявляет `ship`. Элемент `<ejb-relationship-role>` компонента Ship определяет его множественность как `One` и не содержит никаких объявлений `<cmr-field>`, т. к. это однонаправленное отношение.

Двунаправленное отношение «один-ко-многим»

Двунаправленные отношения «один-ко-многим» и «многие-к-одному» – одно и то же, поэтому они рассматриваются в данном разделе вместе. Двунаправленное отношение «один-ко-многим» имеет место тогда, когда один объектный компонент содержит основанное на коллекции поле отношения с другим объектным компонентом, а каждый объектный компонент, содержащийся в этой коллекции, содержит единственную обратную ссылку на содержащий его компонент. Например, в системе «Titan Cruise» каждый компонент Cruise содержит коллекцию ссылок на все заказы билетов, сделанные на данный круиз, а каждый компонент Reservation поддерживает единственную ссылку на свой круиз. Данное отношение с точки зрения компонента Cruise представляет собой двунаправленное отношение «один-ко-многим», а с точки зрения компонента Reservation – двунаправленное отношение «многие-к-одному».

Схема реляционной базы данных

Первая таблица, которая нам потребуется, – это таблица RESERVATION, определенная в следующем листинге. Обратите внимание, что таблица RESERVATION содержит, среди прочего, столбец, выполняющий роль внешнего ключа на таблицу CRUISE:

```
CREATE TABLE RESERVATION
(
    ID INT PRIMARY KEY NOT NULL,
    AMOUNT_PAID DECIMAL (8,2),
    DATE_RESERVED DATE,
    CRUISE_ID INT
}
```

Хотя таблица RESERVATION содержит внешний ключ на таблицу CRUISE, таблица CRUISE не поддерживает обратный внешний ключ на таблицу RESERVATION. Контейнерная система EJB может определить отношение между компонентами Cruise и Reservation, обратившись к таблице RESERVATION, поэтому явные указатели из таблицы CRUISE на таблицу RESERVATION не требуются. Это еще раз иллюстрирует отделение отношений постоянства с точки зрения компонента от фактической реализации этих отношений в базе данных.

Отношение между таблицами RESERVATION и CRUISE показано на рис. 7.13.

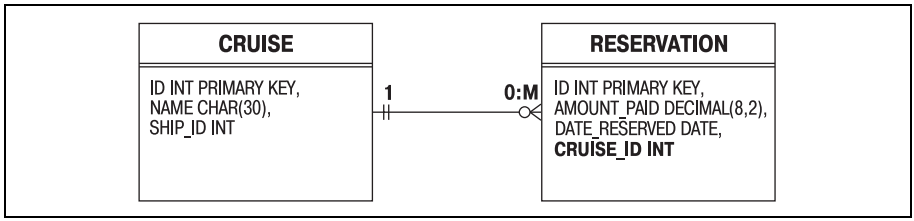

Рис. 7.13. Двунаправленное отношение «один-ко-многим»/«многие-к-одному» в СУРБД

В качестве альтернативы мы могли бы использовать таблицу связей, содержащую внешние ключи к обеим таблицам, RESERVATION и CRUISE. В этой таблице связей, вероятно, на внешний ключ RESERVATION было бы наложено уникальное ограничение, для того чтобы гарантировать, что каждая запись RESERVATION соответствовала бы только одной записи CRUISE.

Абстрактная программная модель

Для того чтобы смоделировать отношение между Cruise и Reservation, прежде всего мы должны определить компонент Reservation, содержащий поле отношения на компонент Cruise:

```
public abstract class ReservationBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(CruiseLocal cruise) {
        return null;
    }
    public void ejbPostCreate(CruiseLocal cruise) {
        setCruise(cruise);
    }

    public abstract void setCruise(CruiseLocal cruise);
    public abstract CruiseLocal getCruise();
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract void setAmountPaid(float amount);
    public abstract float getAmountPaid();
    public abstract void setDate(Date date);
    public abstract Date getDate();

    // Методы обратного вызова
    ...
}
```

Во время создания компонента Reservation в его метод `create()` должна быть передана ссылка на компонент Cruise, для которого он создается. Обратите внимание, что ссылка `CruiseLocal` устанавливается в методе `ejbPostCreate()`, а не в `ejbCreate()`. Как было указано выше, модификация поля отношения не разрешается в методе `ejbCreate()`; это работа метода `ejbPostCreate()`.

Для того чтобы компонент Cruise мог ссылаться на все созданные для него компоненты Reservation, нам необходимо добавить к нему основанное на коллекции поле отношения:

```
public abstract class CruiseBean implements javax.ejb.EntityBean {
    ...

    public abstract void setReservations(Collection res);
    public abstract Collection getReservations();
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract void setName(String name);
    public abstract String getName();
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip();

   // Методы обратного вызова
    ...
}
```

Когда мы создаем отношение между компонентами Cruise и Reservation, зависимость между ними приводит к некоторым любопытным результатам. Например, в процессе создания компонента Reservation происходит его автоматическое добавление к полю отношения компонента Cruise:

```
CruiseLocal cruise = ... get CruiseLocal reference
ReservationLocal reservation = reservationHomeLocal.create( cruise );
Collection collection = cruise.getReservations();
if(collection.contains(reservation))
    // всегда возвращает true
```

Это побочный эффект двунаправленных отношений. Любой компонент Cruise, на который ссылается определенный компонент Reservation, содержит обратную ссылку на этот компонент. Если Reservation X ссылается на Cruise A, то Cruise A должен поддерживать ссылку на Reservation X. Когда мы создаем новый компонент Reservation и устанавливаем ссылку на него в компоненте Cruise, Reservation автоматически добавляется к полю `reservation` компонента Cruise.[1]

Совместное использование ссылок несколькими компонентами приводит к некоторым неприятным последствиям, рассмотренным выше. Например, передача коллекции Reservation от компонента Cruise A компоненту Cruise B фактически перемещает эти отношения в Cruise B, а в Cruise A они больше не существуют (рис. 7.14).

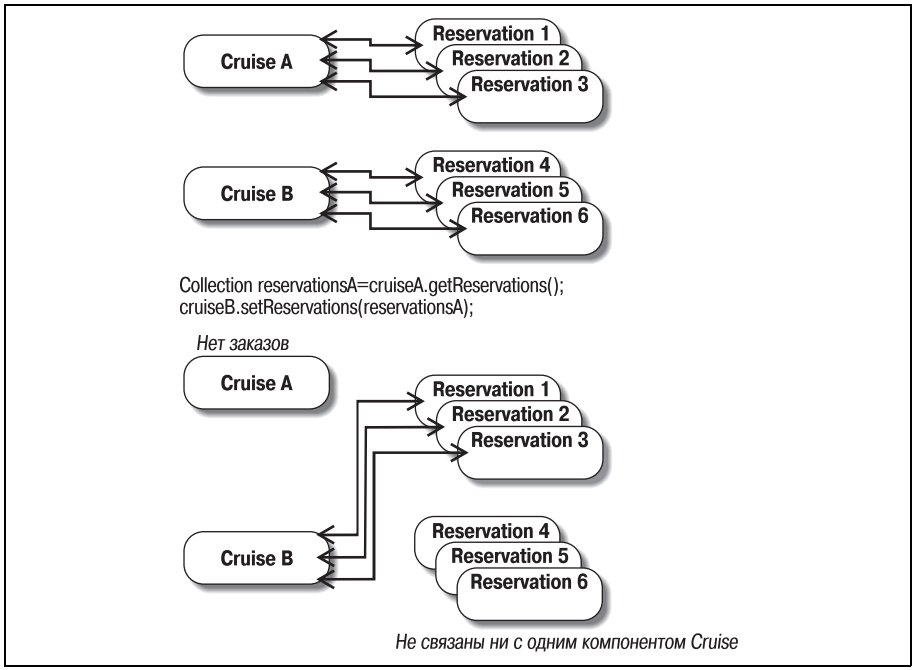

Рис. 7.14. Совместное использование всей коллекции в двунаправленном отношении «один-ко-многим»

[1] В действительности это в большой степени зависит от последовательности действий, контекста транзакции и даже используемого базой данных уровня изоляции. Эти темы подробно освещены в главе 14.

Как и в случае с компонентами Customer и Phone (рис. 7.8), этот эффект обычно нежелателен и его необходимо избежать, поскольку он перемещает набор компонентов Reservation, прежде связанный с компонентом Cruise B.

Можно перемещать всю коллекцию от одного компонента другому и объединять ее с коллекцией второго компонента с помощью метода `Collection.addAll()`, как показано на рис. 7.15.[1] Если переместить коллекцию ссылок компонента Cruise A в компонент Cruise B, то Cruise A больше не будет ссылаться ни на один компонент Reservation, хотя Cruise B будет ссылаться на те, которые были у него перед этим обменом, и те, которые он получил от компонента Cruise A.

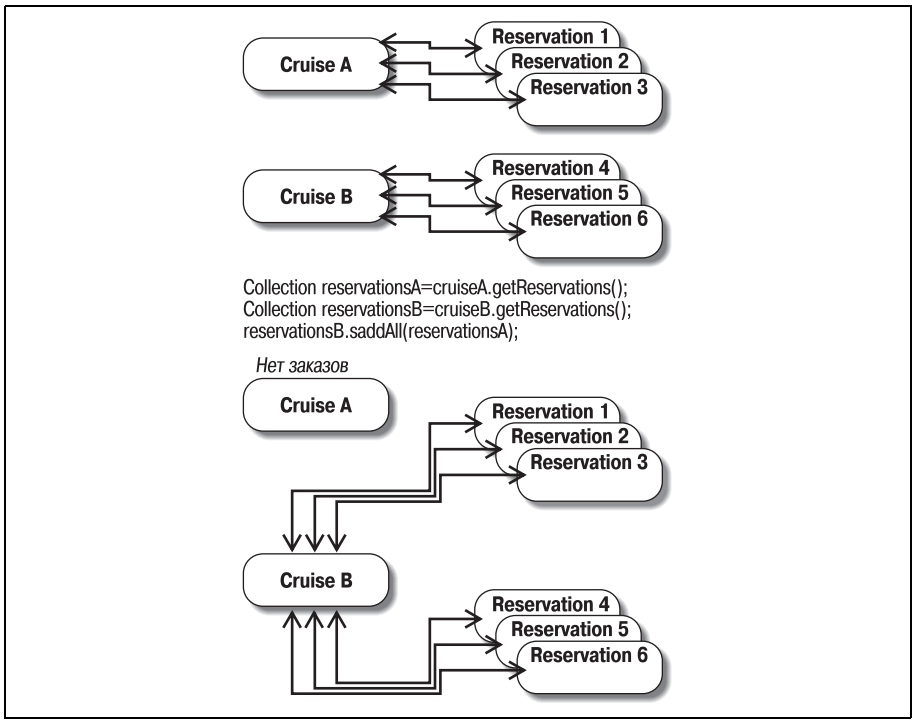

Рис. 7.15. Использование Collection.addAll( ) в двунаправленном отношении «один-ко-многим»

Последствия перемещения отдельного компонента Reservation от одного Cruise другому похожи на те, которые мы наблюдали для других отношений типа «один-ко-многим»: результат такой же, какой был показан на рис. 7.9, когда компонент Phone был перемещен от одного Customer к другому. Интересно отметить, что конечный результат

[1] В EJB 2.0 метод `addAll()` должен поддерживаться полями отношений на базе коллекции.

применения метода `Collection.addAll()` в этом сценарии такой же, как и при использовании метода `Collection.add()` целевой коллекции для каждого элемента исходной коллекции. В обоих случаях каждый элемент исходной коллекции перемещается в целевую коллекцию.

Еще раз напомним, что управляемые контейнером поля отношения, основанные на коллекции, или какие-либо другие должны всегда использовать интерфейс (локальный) `javax.ejb.EJBLocalObject` компонента, а не интерфейс `javax.ejb.EJBObject` (удаленный). Например, была бы ошибкой попытка добавить удаленный интерфейс компонента Reservation (если бы у него такой был) в коллекцию Reservation компонента Cruise. Всякая попытка добавления типа удаленного интерфейса к основанному на коллекции полю отношения приведет к возникновению исключения `java.lang.IllegalArgumentException`.

Абстрактная схема постоянства

В абстрактной схеме постоянства для отношения Cruise–Reservation не вводится никаких новых понятий. Элементы `<ejb-relationship-role>` для обоих компонентов, Cruise и Reservation, содержат элементы `<cmr-field>`. Cruise в качестве своей множественности определяет `One`, а Reservation – `Many`. Далее приведен этот код:

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CruiseEJB</ejb-name>
        <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
        <local>com.titan.cruise.CruiseLocal</local>
        ...
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home>
            com.titan.reservations.ReservationHomeLocal
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        ...
    </entity>
    ...
</enterprise-beans>
<relationships>
    <ejb-relation>
        <ejb-relation-name>Cruise-Reservation
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cruise-has-many-Reservations
            </ejb-relationship-role-name>
```

```
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CruiseEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>reservations</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Reservation-has-a-Cruise
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>ReservationEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>cruise</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

Двунаправленное отношение «многие-ко-многим»

Двунаправленные отношения «многие-ко-многим» имеют место в тех случаях, когда несколько компонентов содержат основанное на коллекции поле отношения с другим компонентом, и каждый компонент, на который ссылается эта коллекция, содержит коллекцию обратных отношений с составными компонентами. Например, в «Titan Cruise» каждый компонент Reservation может ссылаться на несколько Customer (семейный заказ билетов), а у каждого Customer может быть несколько заказов билетов (человек может заказывать билеты несколько раз в год). Это является примером двунаправленных отношений «многие-ко-многим»; клиент отслеживает все свои заказы, а каждый заказ может включать нескольких клиентов.

Схема реляционной базы данных

Таблицы RESERVATION и CUSTOMER уже созданы. Для того чтобы установить двунаправленное отношение «многие-ко-многим», необходимо создать таблицу RESERVATION_CUSTOMER_LINK. Эта таблица содержит два столбца внешних ключей: один для таблицы RESERVATION, а другой для таблицы CUSTOMER:

```
CREATE TABLE RESERVATION_CUSTOMER_LINK
(
    RESERVATION_ID INT,
    CUSTOMER_ID INT
)
```

Отношения между таблицами CUSTOMER, RESERVATION и CUSTOMER_RESERVATION_LINK показаны на рис. 7.16.

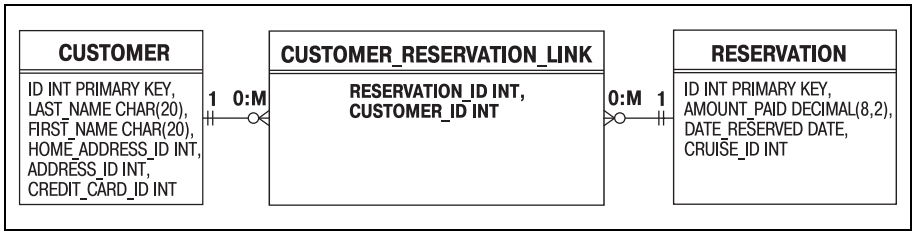

Рис. 7.16. Двунаправленное отношение «многие-ко-многим» в СУРБД

Для реализации двунаправленных отношений типа «многие-ко-многим» в нормализованной реляционной базе данных всегда требуется какая-либо связь.

Абстрактная программная модель

Для моделирования двунаправленного отношения «многие к многим» между компонентами Customer и Reservation мы должны изменить оба класса компонента с целью включения поля коллекции отношений:

```
public abstract class ReservationBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(CruiseLocal cruise,Collection customers) {
        return null;
    }
    public void ejbPostCreate(CruiseLocal cruise,Collection customers) {
        setCruise(cruise);
        Collection myCustomers = this.getCustomers();
        myCustomers.addAll(customers);
    }

    public abstract void setCustomers(Set customers);
    public abstract Set getCustomers();
    ...
}
```

Абстрактные методы доступа, определенные для поля отношений customers объявляют тип коллекции java.util.Set. Тип Set должен содержать только уникальные компоненты Customer без каких-либо повторений. Повторяющиеся компоненты Customer породят в системе резервирования «Титан» некоторые интересные, но нежелательные побочные эффекты. Для того чтобы правильно учитывать количество пассажиров и не допускать переплаты со стороны клиентов, «Титану» необходимо, чтобы Customer был зарегистрирован в одном заказе только один раз. Тип коллекции Set обеспечивает это ограничение. Эффективность типа коллекции Set в значительной степени зависит от

ограничений ссылочной целостности, установленных в конкретной базе данных.

Кроме добавления абстрактных методов доступа `getCustomers()/setCustomers()`, мы изменили методы `ejbCreate()/ejbPostCreate()` так, чтобы они принимали коллекцию компонентов Customer. Во время создания компоненту Reservation должен быть передан список компонентов Customer, которые он добавляет к своей собственной коллекции компонентов Customer. Как и всегда, управляемые контейнером поля отношения не могут изменяться в методе `ejbCreate()`. Изменение управляемых контейнером полей отношения является задачей метода `ejbPostCreate()`, после того как компонент будет создан.

Мы также изменили компонент Customer для того, чтобы он мог содержать коллекцию отношений со всеми своими компонентами Reservation. Хотя мысль, что Customer должен иметь несколько Reservation, может казаться странной, кто-то может захотеть заранее заказать билеты на несколько круизов. Для того чтобы учесть эту возможность, мы расширили компонент Customer с целью включения в него поля отношения `reservation`:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...
    // Поля отношений
    public abstract void setReservations(Collection reservations);

    public abstract Collection getReservations();
    ...
```

Компоненту Reservation во время создания передается ссылка на его компонент Cruise и на коллекцию компонентов Customer. Поскольку отношение определено как двунаправленное, контейнер EJB автоматически добавляет компонент Reservation в поле отношения `reservation` компонента Customer. Следующий фрагмент кода иллюстрирует это:

```
Collection customers = ... get local Customer EJBs
CruiseLocal cruise = ... get a local Cruise EJB
ReservationHomeLocal = ... get local Reservation home

ReservationLocal myReservation = resHome.create(cruise,  customers);

Iterator iterator = customers.iterator();
while(iterator.hasNext()) {
    CustomerLocal customer = (CustomerLocal)iterator.next();
    Collection reservations = customer.getReservations();
    if( reservations.contains( myReservation ))
        // всегда будет возвращать true
}
```

Обмен ссылками на компоненты в двунаправленных отношениях «многие-ко-многим» приводит к настоящему совместному их использова-

нию, где каждое отношение поддерживает ссылку на перемещенную коллекцию. Этот тип отношения показан на рис. 7.17.

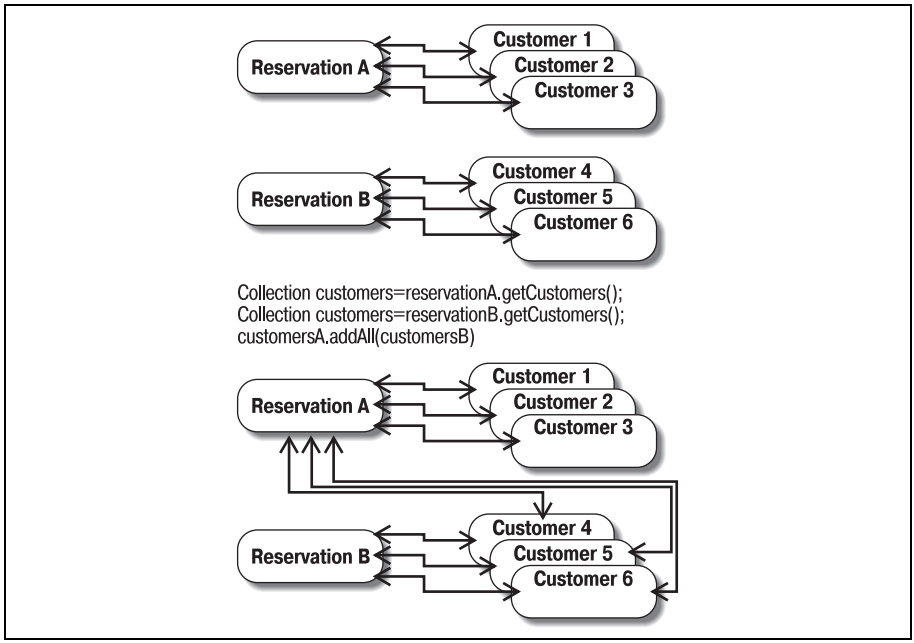

Рис. 7.17. Использование метода Collection.addAll( ) в двунаправленном отношении «многие-ко-многим»

Конечно, применение методов `setCustomers()` или `setReservations()` приведет к обмену ссылками между объектным компонентом и элементами в первоначальной коллекции, но на другие отношения, поддерживаемые этими элементами, оно не повлияет. На рис. 7.18 показано, что может произойти, если в двунаправленном отношении «многие-ко-многим» совместно используется вся коллекция.

После вызова метода `setCustomers()` компонента Reservation D, Customer компонента Reservation D, изменяется на Customer 1, 2, и 3. Перед этой операцией на Customer 1, 2, и 3 ссылался также компонент Reservation A, который сохранил эту ссылку и после ее завершения. Фактически она повлияла только на отношения между компонентами Reservation D и Customer 4, 5, и 6. Отношение между Customer 4, 5, и 6 и другими компонентами Reservation в этой операции не изменились. Это свойство уникально для отношений «многие-ко-многим» (и двунаправленного, и однонаправленного); операции над полями отношения затрагивают только эти специфические отношения, они не влияют ни на одно отношение того же типа с другими компонентами.

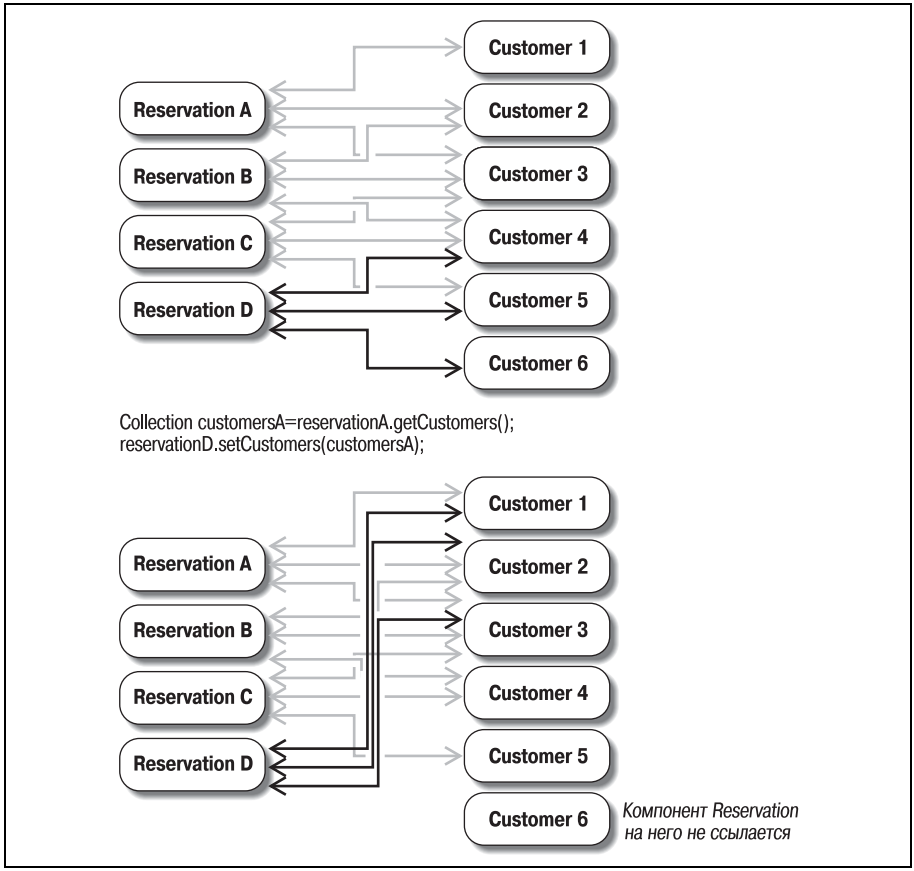

Рис. 7.18. Совместное использование всей коллекции в двунаправленном отношении «многие-ко-многим»

Абстрактная схема постоянства

В абстрактной схеме постоянства двунаправленных отношений «многие-ко-многим» не вводится ничего нового, и она не должна вызвать удивление. Каждый элемент `<ejb-relationship-role>` определяет свою множественность как `Many` и объявляет в `<cmr-field>` определенный тип коллекции:

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CustomerEJB</ejb-name>
        <local-home>com.titan.customer.CustomerHomeLocal</local-home>
        <local>com.titan.customer.CustomerLocal</local>
        ...
```

```
        </entity>
        <entity>
            <ejb-name>ReservationEJB</ejb-name>
            <local-home> com.titan.reservation.ReservationHomeLocal</local-home>
            <local>com.titan.reservation.ReservationLocal</local>
            ...
        </entity>
        ...
    </enterprise-beans>

    <relationships>
        <ejb-relation>
            <ejb-relation-name>Customer-Reservation</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Customer-has-many-Reservations
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>CustomerEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>reservations</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Reservation-has-many-Customers
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>ReservationEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>customers</cmr-field-name>
                    <cmr-field-type>java.util.Set</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
```

Однонаправленное отношение «многие-ко-многим»

Однонаправленные отношения «многие-ко-многим» имеют место, когда несколько компонентов поддерживают коллекцию отношений с другим компонентом, а компонент, на который ссылается коллекция, не содержит обратного отношения с составными компонентами. В системе резервирования билетов «Титан» с каждым заказом (Reservation)

связана каюта (Cabin) на корабле (Ship). Это дает возможность пассажиру (Customer) зарезервировать на корабле (Ship) определенную каюту (Cabin) (например, салон-люкс или каюту среднего класса). В данном случае каждый заказ может быть сделан сразу на несколько кают, т. к. каждый заказ может относиться к нескольким клиентам. Например, семья делает заказ на пять человек в двух смежных каютах (одна – для детей, а другая – для родителей).

Хотя компоненту Reservation необходимо отслеживать объекты Cabin, которые он резервирует, компоненту Cabin не требуется отслеживать все Reservation, созданные всеми компонентами Cruise. Компоненты Reservation ссылаются на коллекцию компонентов Cabin, но компоненты Cabin не содержат обратных ссылок на Reservation.

Схема реляционной базы данных

Наша первая задача состоит в объявлении таблицы CABIN:

```
CREATE TABLE CABIN
(
    ID INT PRIMARY KEY NOT NULL,
    SHIP_ID INT,
    NAME CHAR(10),
    DECK_LEVEL INT,
    BED_COUNT INT
}
```

Обратите внимание, что таблица CABIN содержит внешний ключ на таблицу SHIP. Хотя это отношение и важно, мы здесь не будем его объяснять, т. к. мы в этой главе уже рассматривали двунаправленное отношение «один-ко-многим». Однако для полноты картины отношение Cabin-Ship включено в рис. 7.10. Для того чтобы установить между таблицами RESERVATION и CABIN однонаправленное отношение «многие-ко-многим», нам понадобится таблица RESERVATION_CABIN_LINK:

```
CREATE TABLE RESERVATION_CABIN_LINK
(
    RESERVATION_ID INT,
    CABIN_ID INT
}
```

Отношение между записями в таблицах CABIN и RESERVATION, установленное через таблицу RESERVATION_CABIN_LINK, показано на рис. 7.19.

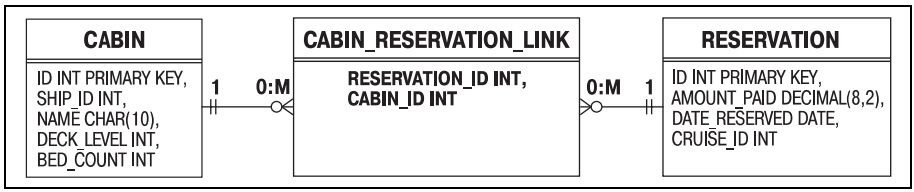

Рис. 7.19. Однонаправленное отношение «многие-ко-многим» в СУРБД

Абстрактная программная модель

Для того чтобы смоделировать это отношение, мы должны добавить к компоненту Reservation поле отношения к компонентам Cabin на базе коллекции:

```
public abstract class ReservationBean implements javax.ejb.EntityBean {
    ...
    public abstract void setCabins(Set cabins);
    public abstract Set getCabins();
    ...
}
```

Кроме этого, нам необходимо определить компонент Cabin. Заметьте, что компонент Cabin не содержит обратного отношения с компонентом Reservation. Отсутствие поля отношения, управляемого контейнером, в компоненте Reservation указывает на то, что данное отношение однонаправленное:

```
public abstract class CabinBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(ShipLocal ship, String name) {
        this.setName(name);
        return null;
    }
    public void ejbPostCreate(ShipLocal ship, String name) {
        this.setShip(ship);
    }
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip();
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract void setName(String name);
    public abstract String getName();
    public abstract void setBedCount(int count);
    public abstract int getBedCount();
    public abstract void setDeckLevel(int level);
    public abstract int getDeckLevel();

    // Методы обратного вызова
}
```

Хотя в компоненте Cabin не определяется поле отношения с компонентом Reservation, в нем определено двунаправленное отношение «один-ко-многим» с компонентом Ship.

Эффект, возникающий при обмене полей отношения в однонаправленном отношении «многие-ко-многим», в основном такой же, как и в двунаправленном отношении «многие-ко-многим». Применение операции `Collection.addAll()` для совместного использования всей коллекции имеет такой же эффект, какой мы видели в предыдущем разделе, посвященном двунаправленным отношениям «многие-ко-многим».

Единственное различие состоит в том, что стрелки направлены только в одну сторону, а не в обе.

Если компонент Reservation удаляет компонент Cabin из своего поля отношения, эта операция не затрагивает другие компоненты Reservation, которые ссылаются на этот компонент Cabin. Это показано на рис. 7.20.

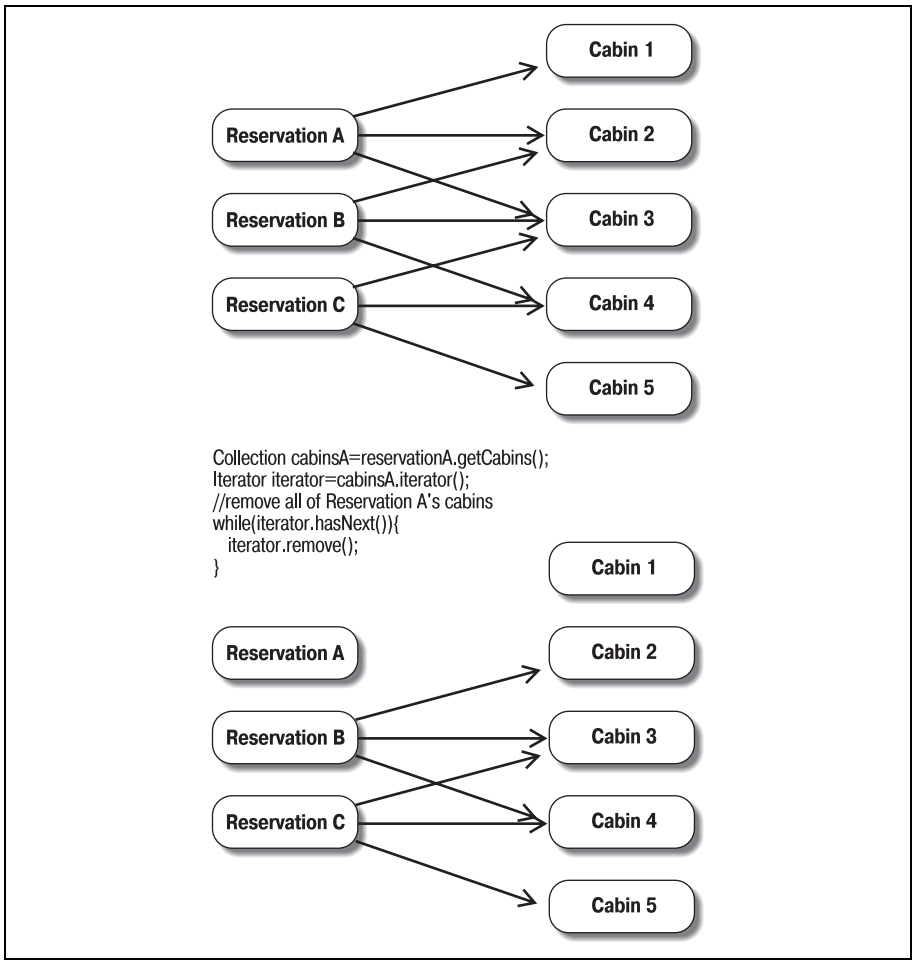

Рис. 7.20. Удаление компонентов в однонаправленном отношении «многие-ко-многим»

Абстрактная схема постоянства

Абстрактная схема постоянства для отношения Reservation–Cabin не содержит никаких сюрпризов. Множественность обоих элементов

<ejb-relationship-role> **указана как** Many**, но элемент** <cmr-field> **определен только в разделе** <ejb-relationship-role> **компонента** Reservation**:**

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CabinEJB</ejb-name>
        <local-home>com.titan.cabin.CabinHomeLocal</local-home>
        <local>com.titan.cabin.CabinLocal</local>
        ...
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home> com.titan.reservation.ReservationHomeLocal</local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        ...
    </entity>
    ...
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Cabin-Reservation</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cabin-has-many-Reservations
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>CabinEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Reservation-has-many-Customers
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>ReservationEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>cabins</cmr-field-name>
                <cmr-field-type>java.util.Set</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

📖 Рабочее упражнение 7.2. Отношения объектов в CMP 2.0: Часть 2

Совмещение компонентов и дескриптор развертывания

Отношения друг с другом могут иметь только объектные компоненты, развернутые вместе в одном дескрипторе развертывания. Объектные компоненты, разворачиваемые вместе, рассматриваются как единый развертываемый модуль или приложение, в котором все объекты используют одну базу данных и расположены в одной JVM. Это ограничение дает возможность контейнерной системе EJB использовать отложенную загрузку, оптимистический параллелизм и другую оптимизацию производительности. Несмотря на то что технически возможно поддерживать отношения между разными приложениями или даже между разными контейнерными системами, сложность такой реализации в сочетании с предсказуемым ухудшением производительности явилась достаточной причиной для того, чтобы ограничить поля отношений теми объектными компонентами, которые развертываются вместе. В будущем отношения объектов могут быть расширены с целью включения удаленных ссылок на объекты, развернутые в других контейнерах или других файлах JAR внутри одного контейнера, но в EJB 2.0 удаленные ссылки не разрешается использовать в качестве типов отношений.

Каскадное удаление и метод Remove

Как мы узнали в главе 5, вызов операции `remove()` для внутреннего или компонентного объекта объектного компонента удаляет данные этого компонента из базы данных. Удаление данных компонента, конечно, влияет на отношения, которые компонент поддерживает с другими объектными компонентами.

При удалении компонента контейнер EJB сначала удаляет его из всех отношений, которые он поддерживает с другими компонентами. Рассмотрим, например, отношение между объектными компонентами, которые мы создали в этой главе (рис. 7.21).

Если бы приложение EJB вызвало метод `remove()` компонента CreditCard, компонент Customer, который ссылается на этот компонент, содержал бы теперь в его поле отношения `creditCard` пустой указатель, как показано в следующем фрагменте кода:

```
CustomerLocal customer = ... get Customer EJB
CreditCardLocal creditCard = customer.getCreditCard();
creditCard.remove();
if(customer.getCreditCard() == null)
    // всегда будет возвращать true
```

В тот момент, когда для локальной ссылки на компонент CreditCard вызывается операция `remove()`, этот компонент отсоединяется от компонента Customer и удаляется. Последствия удаления компонента еще

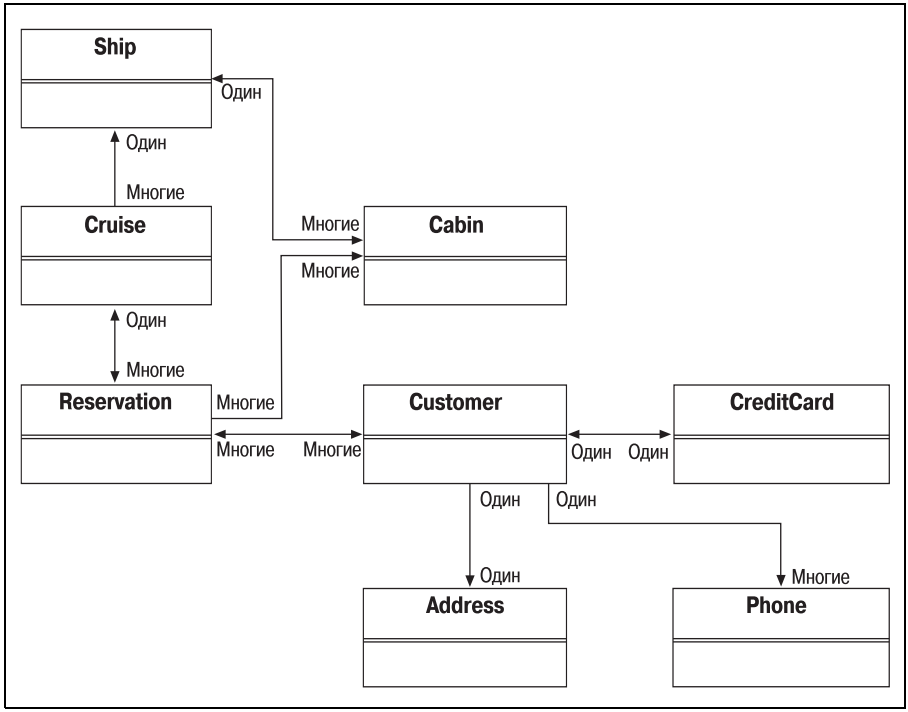

Рис. 7.21. Диаграмма классов «Titan Cruise»

более интересны, если этот компонент участвует в нескольких отношениях. Например, вызов `remove()` компонента Customer повлияет на поля отношений с компонентами Reservation, Address, Phone и CreditCard. При использовании поля отношения с одним компонентом, такого как ссылка на компонент CreditCard в компоненте Customer, поле, соответствующее удаляемому компоненту, будет установлено в значение `null`. В случае с полями отношения на базе коллекции удаляемый объект будет удален из этой коллекции. В некоторых случаях требуется, чтобы удаление объектного компонента вызвало каскадное удаление. Например, если удаляется компонент Customer, необходимо, чтобы компоненты Address, на которые ссылаются его поля отношений `billingAddress` и `homeAddress`, тоже были удалены. Это оградило бы от проблемы несвязанных компонентов Address в базе данных. Элемент `<cascad-delete>` вызывает каскадное удаление; он может использоваться с отношениями «один-к-одному» и «один-ко-многим». Он не имеет смысла в отношениях «многие-ко-многим» и «многие-к-одному» из-за характера этих отношений. Например, в отношении «многие-к-одному» между компонентами Reservation и Cruise отмена заказа билета одним пассажиром не должна отменять сам круиз! Другими словами, мы не хотели бы, чтобы удаление компонента Reservation вызвало стирание его компонента Cruise.

Здесь показано, как изменить объявление отношения для компонентов Customer и Address, чтобы получить каскадное удаление:

```
<relationships>
    <ejb-relation>
        <ejb-relationship-role>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>homeAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <multiplicity>One</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
     </ejb-relation>
</relationships>
```

Если не задать каскадное удаление, запись таблицы ADDRESS, связанная с компонентом Address, не будет удалена при удалении записи CUSTOMER. Это может привести к отсоединенному значению зависимого объекта, т. е. данные ни с чем не будут связаны. В некоторых случаях необходимо указать каскадное удаление, чтобы гарантировать отсутствие отсоединенных объектов после удаления компонента. Однако в других случаях каскадное удаление не требуется. Если, например, запись ADDRESS, связанная с компонентом, используется другими записями CUSTOMER (если, например, два разных клиента проживают по одному адресу), нам, скорее всего, не потребуется удалять его при удалении записи CUSTOMER. Каскадное удаление может быть задано только для объектного компонента, который поддерживает единственную ссылку на удаляемый объект. Например, можно задать каскадное удаление в элементе <ejb-relationship-role> компонента Phone в отношении Customer–Phone при удалении Customer, поскольку на каждый компонент Phone ссылается только один Customer. Однако нельзя задать каскадное удаление в этом отношении для компонента Customer, потому что Customer, возможно, ссылается на несколько компонентов Phone. Объектный компонент, который вызывает каскадное удаление, должен иметь в данном отношении множественность, равную One.

Каскадное удаление влияет только на отношение, для которого оно задано. Так, если вы определяете каскадное удаление для отношения Customer–Phone, но не для отношения Customer–HomeAddress, удаление Customer приведет к удалению всех компонентов Phone, но не

компонентов Address. Для того чтобы удалить их, необходимо также задать каскадное удаление и для компонентов Address.

Каскадное удаление может распространяться по отношениям в цепной реакции. Например, если отношение Ship–Cruise определяет на поле отношения Cruise каскадное удаление, а отношение Cruise–Reservation определяет каскадное удаление на поле отношения Reservation, то при удалении компонента Ship будут удалены все связанные с ним компоненты Cruise и компоненты Reservation для этих Cruise.

Каскадное удаление – мощное средство, но вместе с тем и достаточно опасное, поэтому им следует пользоваться с осторожностью. Эффективность каскадного удаления в значительной степени зависит от ссылочной целостности базы данных. Например, если база данных настроена так, что внешний ключ должен указывать на существующую запись, удаление данных объекта может нарушить это ограничение и вызвать откат транзакции.

📖 Рабочее упражнение 7.3. Каскадное удаление в CMP 2.0

8

EJB 2.0 CMP: EJB QL

Поисковые методы являются частью спецификации Enterprise JavaBeans начиная с версии EJB 1.0. Эти методы определены для локальных и удаленных внутренних интерфейсов объектных компонентов и применяются для их поиска. У всех объектных компонентов должен быть метод `findByPrimaryKey()`, принимающий в качестве параметра первичный ключ компонента и возвращающий ссылку на компонент. Например, компонент Cruise в своем внутреннем интерфейсе определяет стандартный метод поиска первичного ключа следующим образом:

```
public CruiseHomeLocal extends javax.ejb.EJBLocalHome
{
    public Integer create(String name,ShipLocal ship);

    public CruiseLocal findByPrimaryKey(Integer key);

}
```

В дополнение к обязательному методу `findByPrimaryKey()` разработчики объектных компонентов могут определить столько пользовательских поисковых методов, сколько им понадобится. Например, компонент Cruise для поиска круиза с указанным именем может определить метод `findByName()`:

```
public CruiseHomeLocal extends javax.ejb.EJBLocalHome
{
    public Integer create(String name,ShipLocal ship)
```

```
        throws CreateException;

    public CruiseLocal findByPrimaryKey(Integer key)
        throws FinderException;

    public CruiseLocal findByName(String cruiseName)
        throws FinderException;
}
```

Возможность определения пользовательских поисковых методов не является чем-то новым, но до EJB 2.0 не было никакого стандартного способа описания правил функционирования этих поисковых методов. Поведение метода `findByPrimaryKey()` очевидно: поиск объектного компонента с таким же первичным ключом. Однако не всегда это так, поэтому для них требуется дополнительная информация, сообщающая контейнеру, как эти методы должны себя вести. В EJB 1.1 не было никакого стандартного механизма для описания поведения пользовательских поисковых методов, и производителям приходилось изобретать свои собственные языки и методы запросов. Из-за этого пользовательские методы были совершенно не переносимы, и при развертывании системы приходилось гадать, как с их помощью следует выполнять запросы. В EJB 2.0 был введен язык запросов Enterprise JavaBeans (EJB QL, – Enterprise JavaBeans Query Language), представляющий собой стандартный язык запросов для объявления поведения пользовательских поисковых методов и новых методов выборки (select). Методы выборки похожи на поисковые методы, но являются более гибкими и видимы только классу компонента. Поисковым методам и методам выборки в EJB 2.0 дано общее название – методы *запроса* (*query*).

EJB QL – это декларативный язык запросов, похожий на язык структурных запросов (Structured Query Language, SQL), применяемый в реляционных базах данных, но он приспособлен к работе с абстрактной схемой постоянства объектных компонентов в EJB 2.0.

Запросы EJB QL определены в терминах абстрактной схемы постоянства объектных компонентов, а не используемого хранилища данных, поэтому они переносимы между базами данных и схемами данных. Во время развертывания абстрактного класса объектного компонента в контейнере операторы EJB QL обычно просматриваются и транслируются в код доступа к данным, оптимизированный для хранилища данных этого контейнера. Во время выполнения методы запроса, определенные в EJB QL, обычно выполняются на родном языке базового хранилища данных. Например, контейнер, использующий для реализации постоянства реляционную базу данных, может оттранслировать операторы EJB QL в стандартный SQL 92, а контейнер с объектной базой данных мог бы оттранслировать эти же самые операторы в объектный язык запросов.

EJB QL дает возможность разработчикам компонентов описывать поведение методов запроса абстрактным способом, делая запросы переносимыми между различными базами данных и между разными производителями EJB. Язык EJB QL прост для изучения, но вместе с тем он довольно точно переводится в «родной» код базы данных. Это – достаточно мощный и гибкий язык запросов, ускоряющий разработку и, в то же время, быстро выполняющийся в «родном» коде. Однако, как мы увидим дальше, EJB QL не является панацеей и не свободен от недостатков.

Объявление EJB QL

Операторы EJB QL объявляются в элементах <query> дескриптора развертывания объектного компонента. В следующем листинге можно видеть, что у метода findByName(), определенного в локальном внутреннем интерфейсе компонента Cruise, есть свой собственный элемент <query> и оператор EJB QL:

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CruiseEJB</ejb-name>
            ...
            <reentrant>False</reentrant>
            <abstract-schema-name>Cruise</abstract-schema-name>
            <cmp-version>2.x</cmp-version>
            <cmp-field>
                  <field-name>name</field-name>
            </cmp-field>
            <primkey-field>id</primkey-field>
            <query>
                <query-method>
                    <method-name>findByName</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <ejb-ql>
                    SELECT OBJECT(c) FROM Cruise c WHERE c.name = ?1
                </ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

Элемент <query> содержит два основных элемента. Элемент <query-method> указывает поисковый метод удаленного и/или локального внутреннего интерфейса, а элемент <ejb-ql> описывает оператор EJB QL.

А элемент <query> связывает оператор EJB QL с подходящим поисковым методом. Оператор EJB QL будет рассмотрен подробно в начале следующего раздела.

Каждый объектный компонент, упомянутый в операторе EJB QL, должен иметь специальный указатель, называемый *именем абстрактной схемы* (*abstract schema name*), объявляемый элементом <abstract-schema-name>. Элементы <abstract-schema-name> должны иметь уникальные имена; никакие два объектных компонента не могут иметь одно и то же абстрактное имя схемы. В объектном элементе, описывающем компонент Cruise, абстрактное имя схемы объявлено как Cruise. Элемент <ejb-ql> содержит оператор EJB QL, использующий этот идентификатор в своей секции FROM.

В главе 7 мы узнали, что абстрактная схема постоянства объектного компонента задается его элементами <cmp-field> и <cmr-field>. Абстрактное имя схемы также является важной частью абстрактной схемы постоянства. Операторы EJB QL всегда выражаются в терминах абстрактных схем постоянства объектных компонентов. EJB QL использует абстрактные имена схем для идентификации типов объектных компонентов, поля постоянства, управляемого контейнером (Container Managed Persistence, CMP), – для идентификации специфичных данных объектного компонента и управляемые контейнером поля отношений (Container Managed Relationship, CMR) – для создания связей между объектными компонентами.

Методы запроса

Существует два основных типа методов запроса: поисковые методы и методы выборки. Они будут рассмотрены в следующих разделах.

Поисковые методы

Поисковые методы вызываются клиентами EJB (прикладными программами или компонентами) для поиска и получения удаленных или локальных ссылок на определенные объектные компоненты. Например, вызов метода findByPrimaryKey() внутреннего интерфейса компонента Customer позволит получить ссылку на определенный компонент Customer.

Поисковые методы всегда объявляются в локальных и удаленных внутренних интерфейсах объектного компонента. Как вы уже знаете, в каждом внутреннем интерфейсе должен быть определен метод findByPrimaryKey(), он относится к типу однообъектных поисковых методов (single-entity find methods). Указание для поискового метода единственного типа возвращаемого удаленного или локального значения указывает на то, что этот метод выполняет поиск только одного компонента. Ясно, что findByPrimaryKey() возвращает только одну удаленную

ссылку, из-за того что между значением первичного ключа и объектом устанавливается отношение «один-к-одному». Кроме этого могут быть объявлены также и другие поисковые методы для одного объекта. Например, в следующем фрагменте кода компонент Customer определяет несколько поисковых методов для одного объекта, поддерживающих разные варианты запроса:

```
public interface CustomerHomeRemote extends javax.ejb.EJBHome {
    public CustomerRemote findByPrimaryKey(Integer primaryKey)
        throws javax.ejb.FinderException, java.rmi.RemoteException;

    public CustomerRemote findByName(String lastName, String firstName)
        throws javax.ejb.FinderException, java.rmi.RemoteException;

    public CustomerRemote findBySSN(String socialSecurityNumber)
        throws javax.ejb.FinderException, java.rmi.RemoteException;
}
```

Кроме того, разработчики компонентов могут определять многообъектные поисковые методы (multi-entity find methods), возвращающие коллекцию компонентных объектов. В следующем листинге показаны два многообъектных поисковых метода:

```
public interface CustomerHomeLocal extends javax.ejb.EJBLocalHome {
    public CustomerLocal findByPrimaryKey(Integer primaryKey)
        throws javax.ejb.FinderException;

    public Collection findByCity(String city,String state)
        throws javax.ejb.FinderException;

    public Collection findByGoodCredit()
        throws javax.ejb.FinderException;
}
```

Для того чтобы вернуть из поискового метода несколько ссылок, необходимо использовать тип `java.util.Collection`.[1] Поисковый метод, возвращающий значения этого типа, может включать дубликаты. Для удаления дубликатов можно указать в операторе EJB QL, связанном с данным поисковым методом, ключевое слово `DISTINCT`. Эта техника объясняется более подробно далее в этой главе в разделе «Использование DISTINCT». Если не найден ни один подходящий компонент, многообъектные поисковые методы возвращают пустую коллекцию.

Enterprise JavaBeans определяет, что все методы запроса (поисковые или выборки) должны быть объявлены как генерирующие исключение `javax.ejb.FinderException`. Поисковые методы, возвращающие единственную удаленную ссылку, генерируют `FinderException`, если имеет место прикладная ошибка, и `javax.ejb.ObjectNotFoundException`, если не

[1] В CMP 2.0 `java.util.Collection` является единственным типом коллекции, поддерживаемым многообъектными поисковыми методами. EJB 1.1 CMP и EJB 2.0 BMP дополнительно поддерживают тип `java.util.Enumeration`.

найден соответствующий компонент. Исключение `ObjectNotFoundException` является подтипом `FinderException` и генерируется только однообъектными поисковыми методами.

Каждый поисковый метод, объявленный в локальном или удаленном внутреннем интерфейсе CMP 2.0 объектного компонента, должен иметь соответствующее объявление запроса в дескрипторе развертывания компонента. В следующем фрагменте дескриптора развертывания компонента Customer показаны объявления двух поисковых методов, `findByName()` и `findByGoodCredit()`, из более ранних примеров:

```
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName = ?1 AND c.firstName = ?2
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>findByGoodCredit</method-name>
        <method-params/>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.hasGoodCredit = TRUE
    </ejb-ql>
</query>
```

Элементы `<query>` в дескрипторе развертывания позволяют разработчику компонента связывать операторы EJB QL с отдельными поисковыми методами. Во время развертывания компонента контейнер пытается сопоставить поисковый метод, объявленный в каждом из элементов `<query>`, с поисковыми методами во внутренних интерфейсах объектного компонента. Это делается с помощью сравнения значений элементов `<method-name>` и `<method-params>` с именами методов и типами параметров (с учетом их порядка) во внутренних интерфейсах.

Если два поисковых метода в локальном и удаленном внутренних интерфейсах имеют одинаковые имена и параметры, то описание запроса применяется к обоим этим методам. Контейнер вернет правильный тип для каждого метода запроса: удаленный объект возвратит один или несколько удаленных компонентных объектов, а локальный – один или несколько локальных компонентных объектов. Это дает возможность определять поведение и локальных, и удаленных внутрен-

них поисковых методов с помощью единственного элемента <query>, что удобно, если необходимо, чтобы у локальных клиентов был доступ к тем же поисковым методам, что и у удаленных.

Элемент <ejb-ql> определяет оператор EJB QL для отдельного поискового метода. Вы, возможно, заметили, что операторы EJB QL могут принимать входные параметры (например, ?1, ?2, ...?n), соответствующие элементу <method-param> поискового метода, а также литералы (например, TRUE). Использование входных параметров и литералов будет более подробно рассмотрено далее в этой главе.

За исключением методов findByPrimaryKey(), все однообъектные и многообъектные поисковые методы должны быть объявлены в элементах <query> дескриптора развертывания. Объявления запросов для методов findByPrimaryKey() не обязательны и фактически запрещены. Ясно, что должен делать этот метод, и вы не должны пытаться изменить его поведение.

Методы выборки

Методы выборки (select methods) похожи на поисковые методы, но они более универсальны и могут использоваться только внутри класса компонента. Другими словами, методы выборки – это закрытые методы запроса, они не предоставляются клиентам объектного компонента через его внутренние интерфейсы.

Еще одно отличие между методами выборки и поисковыми состоит в контексте транзакции, в котором они выполняются. Метод выборки выполняется в контексте использующих его транзакции прикладной задачи или метода обратного вызова, а поисковые методы выполняются в соответствии с их собственными атрибутами транзакции, заданными поставщиком этого компонента.

Методы выборки объявляются как абстрактные методы, в соответствии с соглашением об именах ejbSelect<*ИМЯ-МЕТОДА*>. В следующем коде показаны четыре метода выборки, объявленные в классе AddressBean:

```
public class AddressBean implements javax.ejb.EntityBean {
    ...
    public abstract String ejbSelectMostPopularCity() throws FinderException;

    public abstract Set ejbSelectZipCodes(String state)
        throws FinderException;

    public abstract Collection ejbSelectAll() throws FinderException;

    public abstract CustomerLocal ejbSelectCustomer(AddressLocal addr)
        throws FinderException;
    ...
```

Методы выборки могут возвращать значения полей CMP. Метод ejbSelectMostPopularCity(), например, возвращает единственное значение ти-

па String – название города, на который ссылается большинство компонентов Address.

Для того чтобы вернуть из метода выборки несколько ссылок, необходимо объявить, что тип возвращаемого значения будет либо java.util.Collection, либо java.util.Set.[1] Метод выборки, использующий тип Set, не будет возвращать повторяющиеся значения, тогда как тип Collection может содержать дубликаты. Многообъектные методы выборки возвращают пустой объект Collection или Set, если не найден ни один подходящий компонент. Метод ejbSelectZipCodes() возвращает набор строковых значений – уникальную коллекцию всех почтовых индексов, объявленных для компонентов Address определенного штата.

Как и в поисковых методах, в методах выборки могут быть объявлены несколько параметров, позволяющих ограничить область запроса. Оба метода: ejbSelectZipCodes() и ejbSelectCustomer(), объявляют аргументы, ограничивающие количество возвращаемых результатов. Эти аргументы применяются в качестве входных параметров в операторах EJB QL, связанных с этими методами выборки.

Методы выборки могут возвращать локальные и удаленные компонентные объекты. Для однообъектных методов выборки этот тип определяется типом возвращаемого значения метода ejbSelect(). Например, метод ejbSelectCustomer() возвращает локальный компонентный объект – CustomerLocal. Этот метод легко мог бы быть определен для возвращения удаленного компонентного объекта, с помощью смены типа возвращаемого значения на тип удаленного интерфейса компонента Customer (CustomerRemote). Многообъектные методы выборки, возвращающие коллекцию компонентных объектов по умолчанию возвращают локальные компонентные объекты. Однако поставщик компонента может изменить это заданное по умолчанию поведение, указав в элементе <query> метода выборки специальный элемент <result-type-mapping>.

В следующем фрагменте XML-дескриптора развертывания объявлены два метода выборки из предыдущего примера. Обратите внимание, что они точно такие же, как и объявления поисковых методов. Поисковые методы и методы выборки объявляются в одной части дескриптора развертывания, внутри элемента <query>, вложенного в элемент <entity>:

```
<query>
    <query-method>
        <method-name>ejbSelectZipCodes</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
```

[1] В будущих версиях могут быть добавлены другие типы коллекций, такие как java.util.List и java.util.Map.

```
            SELECT a.homeAddress.zip FROM Address AS a
            WHERE a.homeAddress.state = ?1
        </ejb-ql>
    </query>
    <query>
        <query-method>
            <method-name>ejbSelectAll</method-name>
            <method-params/>
        </query-method>
        <result-type-mapping>Remote</result-type-mapping>
        <ejb-ql>
            SELECT OBJECT(a) FROM Address AS a
        </ejb-ql>
    </query>
```

Имя, заданное в каждом элементе `<method-name>`, должно соответствовать одному из методов `ejbSelect`*`<ИМЯ-МЕТОДА>`*`()`, определенных в классе компонента. Это отличается от поисковых методов CMP 2.0 компонента, у которых в классе компонента нет соответствующих методов `ejbFind`. Для поисковых методов мы указываем имя метода в локальном или удаленном внутреннем интерфейсе. С другой стороны, методы выборки не объявляются ни в локальном, ни в удаленном внутреннем интерфейсе, поэтому мы и используем в классе компонента метод с именем `ejbSelect()`.

Методы выборки могут возвращать локальные или удаленные компонентные объекты. По умолчанию и однообъектные, и многообъектные методы возвращают типы локальных компонентных объектов. Однако поставщик компонента может изменить это заданное по умолчанию поведение с помощью специального элемента `<result-type-mapping>` в элементе `<query>` метода выборки. Значением элемента `<result-type-mapping>` может быть либо `Remote`, либо `Local`. Значение `Local` указывает на то, что метод выборки должен возвращать локальные компонентные объекты, а `Remote` определяет удаленные компонентные объекты. Если элемент `<result-type-mapping>` не задан, в качестве значения по умолчанию используется `Local`. Для однообъектного выбора действительный тип возвращаемого значения метода `ejbSelect()` должен совпадать с указанным в `<result-type-mapping>`. Например, если однообъектный метод `ejbSelect()` возвращает тип `EJBObject`, то `<result-type-mapping>` должен иметь значение `Remote`. В предыдущем примере `<result-type-mapping>` в элементе `<query>` для метода `ejbSelectAll()` был объявлен как `Remote`, а это означает, что запрос должен возвращать типы удаленного компонентного объекта (т. е. удаленные ссылки на компонент Address).[1]

Методы выборки не ограничены контекстом какого-либо отдельного объектного компонента. Они могут применяться с целью выполнения

[1] Это только иллюстрация. Нежелательно (хотя и возможно), чтобы вы как разработчик определяли для компонента Address удаленный интерфейс, поскольку он слишком мал для использования удаленными клиентами.

запроса для всех объектных компонентов, объявленных в том же дескрипторе развертывания. Методы выборки могут вызываться классом компонента из его методов `ejbHome()`, из любых прикладных методов или из методов `ejbLoad()` и `ejbStore()`. В большинстве случаев методы выборки будут вызываться из `ejbHome()` или прикладных методов класса компонента. Методы `ejbHome()`, `ejbLoad()` и `ejbStore()` рассматриваются более подробно в главе 11.

Самый важный момент, относящийся к методам выборки, который необходимо запомнить, состоит в том, что хотя они и могут делать все, что могут поисковые методы, и даже больше, но могут использоваться только классом объявившего их компонента, а не клиентами этого компонента.

Примеры EJB QL

EJB QL выражается в терминах абстрактной схемы постоянства объектного компонента – его абстрактным именем схемы, полями CMP и полями CMR. EJB QL использует абстрактные имена схемы для идентификации компонентов, поля CMP – для задания значений и поля CMR – для навигации по отношениям.

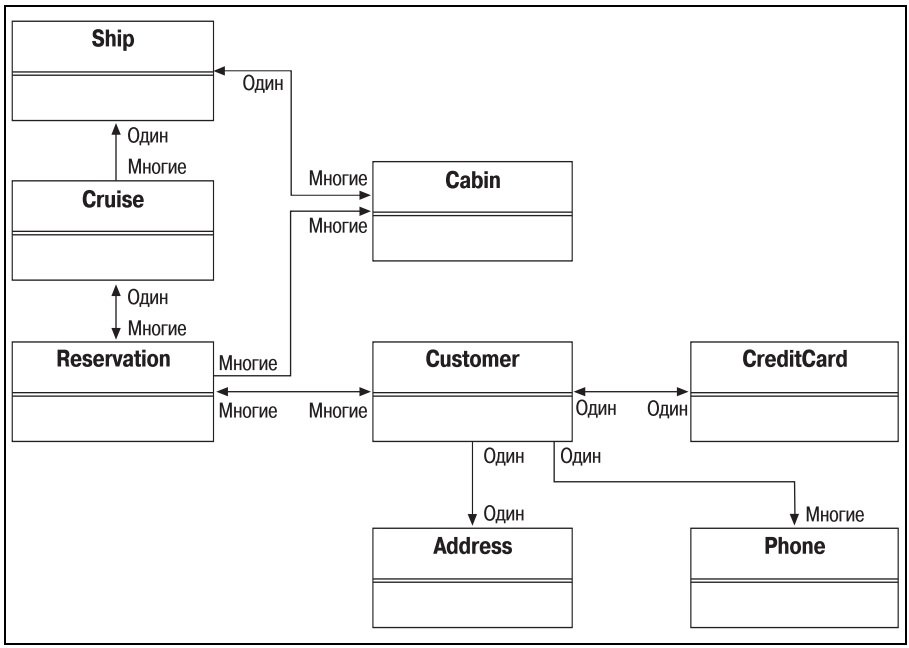

Рис. 8.1. Диаграмма классов «Titan Cruise»

Обсуждая EJB QL, мы будет использовать отношения между компонентами Customer, Address, CreditCard, Cruise, Ship, Reservation и Cabin, созданными в главе 7. На рис. 8.1 изображена диаграмма классов,

показывающая направления и множественность отношений между этими компонентами.

Простые запросы

У простейшего оператора EJB QL нет секции WHERE, а есть лишь абстрактный тип схемы. Например, мы могли бы определить метод запроса для выбора всех компонентов Customer:

```
SELECT OBJECT( c ) FROM Customer AS c
```

Секция FROM определяет, какие типы объектных компонентов будут включены в оператор выбора, т. е. определяет *зону* (*scope*) выбора. В данном случае секция FROM объявляет, что это будет тип Customer, который является абстрактным именем схемы компонента Customer. Фрагмент AS c этой секции назначает c в качестве идентификатора компонента Customer. Нечто похожее имеет место в SQL, который позволяет связывать идентификатор с таблицей. Идентификаторы могут иметь любую длину и следовать тем же правилами, которые применяются к именам полей в языке программирования Java. Однако идентификаторы не могут совпадать с существующими значениями элементов <ejb-name> и <abstract-schema-name>. Кроме того, имена идентификационной переменной *не* чувствительны к регистру, поэтому идентификатор customer стал бы конфликтовать с абстрактным именем схемы Customer. Например, следующий оператор ошибочен, поскольку Customer – это абстрактное имя схемы компонента Customer:

```
SELECT OBJECT( customer ) FROM Customer AS customer
```

Оператор AS необязательный, но он используется в данной книге, чтобы сделать операторы EJB QL более понятными. Следующие два оператора эквивалентны:

```
SELECT OBJECT(c) FROM Customer AS c

SELECT OBJECT(c) FROM Customer c
```

Секция SELECT определяет тип возвращаемых значений. В данном случае это – компонент Customer, как указано идентификатором c.

Если типом SELECT является одиночный идентификатор объектного компонента, то необходимо использовать оператор OBJECT(). Причина этого требования довольно неопределенна (и, по мнению автора, спецификация без него была бы только лучше), но этот оператор требуется всякий раз, когда типом SELECT является идентификатор объектного компонента. Оператор OBJECT() не используется, если тип SELECT выражен с помощью *пути* (*path*), рассмотренного далее.

Простые запросы с путями

EJB QL допускает, чтобы секция SELECT возвращала все поля CMP или одно поле CMR. Например, мы можем определить простой оператор

выбора, возвращающий фамилии всех клиентов «Титана», следующим образом:

```
SELECT c.lastName FROM Customer AS c
```

Секция SELECT реализует простой путь для выбора в качестве типа возвращаемого значения поля CMP lastName компонента Customer. EJB QL использует имена полей CMP и CMR, объявленные в элементах дескриптора развертывания <cmp-field> и <cmr-field>. Эта нотация опирается на синтаксис языка программирования Java, а именно оператор «точка» (.). Например, предыдущий оператор EJB QL составлен на основе дескриптора развертывания компонента Customer:

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
            <ejb-class>com.titan.customer.CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <abstract-schema-name>Customer</abstract-schema-name>
            <cmp-version>2.x</cmp-version>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
```

Вы также можете использовать типы полей CMR в простых операторах выбора. Например, следующий оператор EJB QL выбирает все компоненты CreditCard из всех компонентов Customer:

```
SELECT c.creditCard FROM Customer c
```

В этом случае оператор EJB QL использует путь для того, чтобы перейти от компонентов Customer к их полям отношений creditCard. Идентификатор creditCard получен из имени <cmr-field>, указанного в элементе, описывающем отношение Customer–CreditCard:

```
<enterprise-beans>
    <entity>
        <ejb-name>CustomerEJB</ejb-name>
        ...
        <abstract-schema-name>Customer</abstract-schema-name>
    </entity>
</enterprise-beans>
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-CreditCard</ejb-relation-name>
```

```
<ejb-relationship-role>
    <ejb-relationship-role-name>
        Customer-has-a-CreditCard
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <ejb-name>CustomerEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <cmr-field-name>creditCard</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
<ejb-relationship-role>
...
```

Пути следует применять тогда, когда они необходимы. Обычно они используются для перемещения по одному или нескольким полям CMR и заканчиваются либо на поле CMR, либо на поле CMP. Например, следующий оператор EJB QL выбирает все поля CMP `city` для всех компонентов Address в каждом компоненте Customer:

```
SELECT c.homeAddress.city FROM Customer c
```

В этом случае в пути указано абстрактное имя схемы компонента Customer – поле CMR `homeAddress` компонента Customer и поля CMP `city` компонента Address. Использование путей в EJB QL похоже на передвижение по объектным ссылкам в языке Java.

Для того чтобы проиллюстрировать более сложные пути, нам придется расширить диаграмму классов. На рис. 8.2 показано, что компонент CreditCard связан с компонентом CreditCompany, имеющим свой собственный компонент Address.

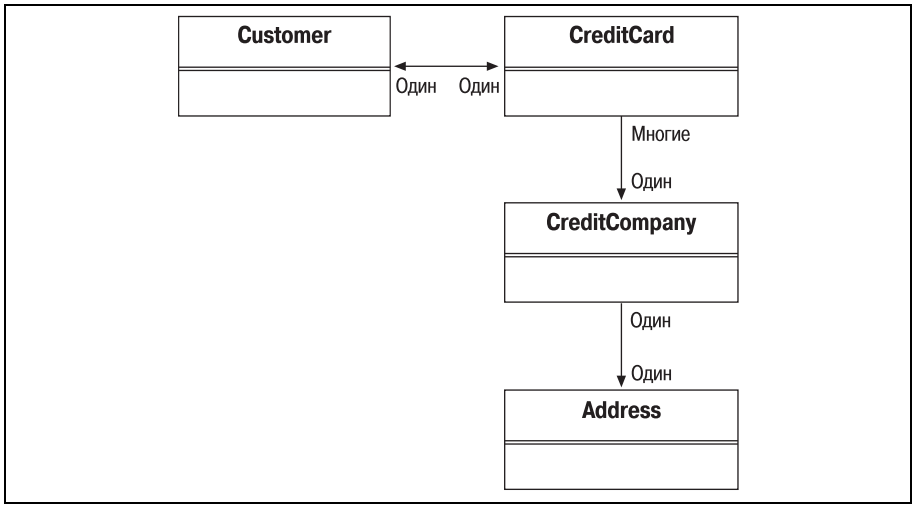

Рис. 8.2. Расширенная диаграмма класса компонента CreditCard

С помощью этих отношений мы можем указать более сложный путь от компонента Customer через компонент CreditCompany к компоненту Address. Следующий оператор EJB QL выбирает все адреса всех компаний, работающих с кредитными картами:

```
SELECT c.creditCard.creditCompany.address FROM Customer AS c
```

Оператор EJB QL также может передвигаться по всем полям CMP компонента Address. Например, следующий оператор выбирает все города, в которых расположены компании, распространяющие кредитные карточки, используемые клиентами «Титана»:

```
SELECT c.creditCard.creditCompany.address.city FROM Customer AS c
```

Обратите внимание, что эти операторы EJB QL возвращают поля CMR `address` и поля CMP `city` только для компаний, отвечающих за кредитные карточки, принадлежащие клиентам «Титана». Информация об адресах компаний, карточки которых в настоящее время не используются клиентами «Титана», не будет присутствовать в результатах запроса.

Пути не могут указывать за пределы полей CMP. Представьте себе, что компонент Address использует класс `ZipCode` в качестве своего поля CMP `zip`:

```
public class ZipCode implements java.io.Serializable {
    public int mainCode;
    public int codeSuffix;
    ...
}
```

Было бы ошибкой пытаться указать одно из полей экземпляра класса `ZipCode`:

```
// Неправильно
SELECT c.homeAddress.zip.mainCode FROM Customer AS c
```

Поля CMP не могут быть разложены и помещены в пути. Все поля CMP считаются монолитными.

Пути, указываемые в секциях `SELECT` операторов EJB QL, всегда должны заканчиваться одним типом. Они не могут оканчиваться полем отношения на базе коллекции. Например, следующий код будет ошибочным из-за поля CMR `reservations`, основанного на коллекции:

```
// Неправильно
SELECT c.reservations FROM Customer AS c
```

Фактически передвижение по основанному на коллекции полю отношения запрещено. Следующий оператор EJB QL также ошибочен, даже несмотря на то, что путь заканчивается полем отношения с одним типом:

```
SELECT c.reservations.cruise FROM Customer AS c
```

Задумавшись над этим, вы увидите, что данное ограничение имеет смысл. В Java тоже нельзя использовать оператор (.) для обращения к элементам java.util.Collection. Например, нельзя делать следующее (предположим, что getReservations() возвращает тип java.util.Collection):

```
// В языке Java это недопустимо
customer.getReservations().getCruise();
```

Ссылка на элементы основанного на коллекции поля отношения в EJB QL возможна, но она потребует применения оператора IN и идентификационного назначения в секции FROM, которые будут рассматриваться далее.

Оператор IN

Многие отношения между объектными компонентами строятся на базе коллекций, поэтому возможность доступа и выбора из этих отношений очень важна. Мы уже видели, что нельзя выбирать элементы непосредственно из отношения, основанного на коллекции. Для того чтобы обойти это ограничение, в EJB QL введен оператор IN, позволяющий идентификатору представлять отдельные элементы в основанном на коллекции поле отношения.

Следующий запрос для выбора элементов из основанного на коллекции отношения включает оператор IN. Он возвращает все заказы билетов для всех клиентов:

```
SELECT OBJECT( r )
FROM Customer AS c,  IN( c.reservations ) AS r
```

Оператор IN назначает индивидуальные элементы в поле CMR reservations идентификатору r. Если у нас есть идентификатор, представляющий индивидуальные элементы коллекции, то мы можем ссылаться на них и даже выбирать их непосредственно в операторе EJB QL. Мы можем также использовать идентификатор элемента при составлении пути. Например, следующий оператор EJB QL выбирает все круизы, на которые клиенты заказали билеты:

```
SELECT r.cruise
FROM Customer AS c, IN( c.reservations ) AS r
```

Идентификаторы, назначенные в секции FROM, вычисляются слева направо. Если идентификатор объявлен, то это можно использовать в последующих объявлениях в секции FROM. Обратите внимание, что идентификатор c, объявленный первым, впоследствии применяется в операторе IN для определения идентификатора r.

Оператор OBJECT() применяется для одиночных идентификаторов в операторе SELECT, а не для определения пути. Хотя это соглашение и не имеет большого смысла, тем не менее оно требуется спецификацией EJB 2.0. Как правило, если типом выбора является одиночный идентификатор объектного компонента, он должен быть заключен в оператор OBJECT(). Если тип выбора – выражение пути, это не обязательно.

Идентифицирующие цепочки, в которых последующая идентификация зависит от предыдущей, могут стать очень длинными. В следующей инструкции EJB QL с помощью двух операторов IN организовано перемещение по двум коллекциям отношений и одному отношению CMR. Хотя этот оператор и не очень полезен, он демонстрирует, как запрос может использовать операторы IN для нескольких отношений:

```
SELECT cbn.ship
FROM Customer AS c, IN ( c.reservations ) AS r,
IN( r.cabins ) AS cbn
```

Рабочее упражнение 8.1. Простые операторы EJB QL

Использование DISTINCT

Ключевое слово DISTINCT гарантирует, что запрос не вернет повторяющиеся значения. Оно особенно полезно в тех случаях, когда применяется к операторам EJB QL, используемым поисковыми методами. Поисковые методы в CMP 2.0 поддерживают только один тип возвращаемого значения – java.util.Collection, способный возвращать дубликаты. Однако в случае применения ключевого слова DISTINCT результаты запроса поискового метода не будут содержать дубликатов.

Так, следующий поисковый метод и связанный с ним запрос возвращают дубликаты:

```
// этот поисковый метод определен в удаленном или локальном внутреннем
интерфейсе
public java.util.Collection findAllCustomersWithReservations()

// этот оператор EJB QL связан с поисковым методом
SELECT OBJECT( cust ) FROM Reservation res, IN (res.customers) cust
```

Предыдущий поисковый метод и связанный с ним оператор EJB QL возвращают все компоненты Customer, имеющие заказы, но если у компонента Customer есть несколько заказов, то итоговая коллекция будет содержать повторяющиеся ссылки на этот компонент. Наличие ключевого слова DISTINCT гарантирует, что в коллекции каждый компонент Customer будет представлен только один раз:

```
SELECT DISTINCT OBJECT( cust ) FROM Reservation res,
IN (res.customers) cust
```

Ключевое слово DISTINCT также может применяться и с методами выборки. Для методов выборки, имеющих тип возвращаемого значения Collection, результат его применения будет таким же. Но если тип возвращаемого значения метода выборки – java.util.Set, то результат не будет содержать дубликатов, вне зависимости от того, указано ключевое слово DISTINCT или нет.

Тип Set явно определен как не содержащий дубликаты. Применение типа Set для возвращаемых значений в комбинации с ключевым словом DISTINCT избыточно, но ошибкой не является.

Секция WHERE и литералы

Для сужения области выбранных элементов в EJB QL также могут применяться литеральные выражения. Это делается в секции WHERE, работающей так же, как и соответствующая секция WHERE языка SQL.

Например, можно создать оператор EJB QL для выбора всех компонентов Customer, использующих определенный тип кредитной карточки. Литерал в этом случае – строковый. Литерные строки заключаются в одинарные кавычки. В литерных значениях, содержащих одинарную кавычку, таких как имя ресторана «Wendy's», необходимо наличие двух одинарных кавычек: 'Wendy''s'. Следующий оператор вернет всех клиентов, использующих кредитную карточку «American Express»:

```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.creditCard.organization = 'American Express'
```

Выражения для пути всегда располагаются в секции WHERE, точно так же, как и в секции SELECT. При выполнении сравнения с литералом выражение пути должно вычисляться для поля CMP. Нельзя сравнивать поле CMR с литералом.

Кроме строковых литералы могут иметь точные (типы long) и вещественные (типы double) числовые значения. Точные числовые литеральные значения выражаются при помощи синтаксиса Java для целых литералов (321, -8932, +22). Значения для вещественных числовых литералов выражаются при помощи синтаксиса Java для литералов с плавающей точкой в научной (5E3, -8.932E5) или десятичной (5.234, 38282.2) нотации.

Например, следующий оператор EJB QL выбирает все суда с водоизмещением 100 000 тонн:

```
SELECT OBJECT( s )
FROM Ship AS s
WHERE s.tonnage = 100000.00
```

Булевы литералы принимают значения TRUE и FALSE. Здесь приведен оператор EJB QL, выбирающий всех клиентов, имеющих ненулевой счет:

```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.hasGoodCredit = TRUE
```

Секция WHERE и входные параметры

Методы запроса (поисковые методы и методы выборки), в которых применяются операторы EJB QL, могут определять аргументы метода. *Входные параметры* (*input parameters*) позволяют сопоставить эти аргументы с операторами EJB QL и применяются для сужения области запроса. Например, метод ejbSelectByCity() предназначен для выбора всех клиентов, постоянно находящихся в определенном городе и штате:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...
    public abstract Collection ejbSelectByCity(String city,String state)
        throws FinderException;
    ...
}
```

В операторе EJB QL для этого метода в качестве входных параметров могут выступать аргументы city и state:

```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.homeAddress.state = ?2
AND c.homeAddress.city = ?1
```

Входные параметры начинаются префиксом ?, за которым следует позиция аргумента в методе запроса. В данном случае city представляет собой первый аргумент, указанный в методе ejbSelectByCity(), а state – второй. Когда метод запроса объявляет один или несколько аргументов, то связанный с ним оператор EJB QL может использовать их (все или только часть) в качестве своих входных параметров.

Входные параметры не ограничиваются простыми типами полей CMP, они также могут быть ссылками на компонентный объект. Например, в локальном интерфейсе компонента Cruise объявлен следующий поисковый метод findByShip():

```
public interface CruiseLocal extends javax.ejb.EJBLocalObject {
    public Collection findByShip( ShipLocal ship )
        throws FinderException;
}
```

В операторе EJB QL, связанном с этим методом, параметр ship мог бы применяться для нахождения всех круизов, запланированных для указанного компонента Ship:

```
SELECT OBJECT( crs ) FROM Cruise AS crs
WHERE crs.ship = ?1
```

Когда в качестве входного параметра выступает компонентный объект, контейнер выполняет сравнение по его первичному ключу. В данном случае он просматривает все компоненты Cruise в поиске ссылки на компонент Ship с таким же первичным ключом, как и у переданного методу запроса компонента Ship.

Секция WHERE и приоритет операций

Секция WHERE состоит из условных выражений, уменьшающих область запроса и ограничивающих количество выбранных элементов. В выражениях могут присутствовать несколько условных и логических операторов. Они перечислены ниже в порядке уменьшения приоритета. Операторы, расположенные в верхней части списка, имеют самый высокий приоритет и вычисляются первыми:

1. Оператор «точка» (.)
2. Арифметические операторы:

   +, – унарные

   *, / умножение и деление

   +, – сложение и вычитание
3. операторы сравнения:

   =, >, > =, <, < =, < > (не равно)

   LIKE, BETWEEN, IN, IS NULL, IS EMPTY, MEMBER OF
4. Логические операторы:

   NOT, AND, OR

Секция WHERE и разделы CDATA

Операторы EJB QL объявляются в XML-дескрипторах развертывания. В качестве разделителей тегов XML применяются символы «больше» (>) и «меньше» (<), поэтому наличие этих символов в операторах EJB QL приведет к ошибкам синтаксического анализа, если не использовать разделы CDATA. Например, следующий оператор EJB QL вызывает ошибку при синтаксическом разборе, поскольку синтаксический анализатор XML интерпретирует символ > как не на месте расположенный разделитель тега XML:

```
<query>
    <query-method>
        <method-name>findWithPaymentGreaterThan</method-name>
        <method-params>
            <method-param>java.lang.Double</method-param>
        </method-params>
```

```
    </query-method>
    <ejb-ql>
        SELECT OBJECT( r ) FROM Reservation r
        WHERE r.amountPaid > ?1
    </ejb-ql>
</query>
```

Для того чтобы обойти эту проблему, необходимо поместить оператор EJB QL в раздел CDATA:

```
<query>
    <query-method>
        <method-name>findWithPaymentGreaterThan</method-name>
        <method-params>
            <method-param>java.lang.Double</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        <![CDATA[
        SELECT OBJECT( r ) FROM Reservation r
        WHERE r.amountPaid  > 300.00
        ]]>
    </ejb-ql>
</query>
```

Раздел CDATA имеет вид <![CDATA[*литерный-текст*]]>. Когда XML-процессор доходит до раздела CDATA, он не пытается анализировать содержимое, заключенное в разделе CDATA, а обрабатывает его как литеральный текст.[1]

Секция WHERE и арифметические операторы

Посредством арифметических операторов запрос в процессе выполнения сравнения может вычислять арифметические выражения. Арифметические операторы в EJB QL могут применяться только в секции WHERE, а не в секции SELECT.

Следующий оператор EJB QL возвращает ссылки на все компоненты Reservation, с которых будет взиматься портовый налог, превышающий 300 долларов:

```
SELECT OBJECT( r ) FROM Reservation r
WHERE (r.amountPaid * .01)  > 300.00
```

К арифметическим операциям применяются такие же правила, какие действуют в языке программирования Java, где числа в процессе вы-

[1] Об XML и разделах CDATA подробно рассказано в книге «XML. Справочник», Элиотт Расти Гарольд и У. Скотт Минс, пер. с англ., изд-во «Символ-Плюс» 2002 г. («XML in Nutshell» Elliotte Rusty Harold & W. Scott Means O'Rielly & Associates, 2001).

полнения вычислений *расширяются* (*widened*), или *повышаются* (*promoted*). Например, умножение значений с типами `double` и `int` требует, чтобы значение `int` сначала было повышено до значения `double`. (Тип результата всегда совпадает со значением самого широкого типа, используемого в вычислении, поэтому умножение `int` и `double` дает результат `double`.)

Типы `String`, `boolean` и тип компонентного объекта не могут участвовать в арифметических операциях. Например, операция сложения двух значений типа `String` считается ошибочной. Есть специальная функция для сложения (конкатенации) значений типа `String`, рассмотренная далее в разделе «Секция WHERE и функциональные выражения».

Секция WHERE и логические операторы

Логические операторы, такие как `AND`, `OR` и `NOT`, работают в EJB QL таким же образом, как и соответствующие им логические операторы SQL.

Логические операторы вычисляют только выражения `Boolean`, поэтому каждый операнд (каждая из сторон выражения) должен быть приведен к значению true или false. Логические операторы имеют самый низкий приоритет для того, чтобы до их применения были вычислены все выражения.

Однако операторы `AND` и `OR` не могут вести себя так же, как их аналоги в языке Java – операторы `&&` и `||`. EJB QL не определяет, будут ли условно вычисляться правые операнды. Например, оператор `&&` в Java вычисляет свой правый операнд, только если левый имеет значение true. Точно так же, логический оператор `||` вычисляет правый операнд, только если левый операнд принимает значение false. Для операторов `AND` и `OR` в EJB QL мы не можем делать таких предположений. Вычисляют ли эти операторы правые операнды, зависит от родного языка запросов, в который эти операторы будут оттранслированы. Лучше предположить, что во всех логических операторах будут вычисляться оба операнда.

`NOT` просто инвертирует значение `Boolean` своего операнда. Выражения, имеющие булево значение true, принимают значение false и наоборот.

Секция WHERE и знаки сравнения

Операторы сравнения, в которых используются символы =, >, > =, <, < = и < >, должны быть вам знакомы. Следующий оператор выбирает все компоненты Ship, в которых содержимое поля CMP `tonnage` больше или равно 80 000, но меньше или равно 130 000 (тоннам):

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage >= 80000.00 AND s.tonnage <= 130000.00
```

Со значениями типа String, boolean и ссылками на компонентный объект могут применяться только операторы = и < > (не равно). Символы «больше» и «меньше» (> >, =, <, < =) могут использоваться только для числовых значений. Например, сравнение двух строк при помощи символов «больше» и «меньше» было бы ошибкой. В EJB QL механизм такого сравнения строк не реализован.

Секция WHERE и семантика равенства

Хотя и можно сравнивать точные числовые значения (short, int, long) с вещественными значениями (double, float), но во всех других сравнениях на равенство должны участвовать одинаковые типы. К примеру, нельзя сравнивать строковое значение 123 с литералом 123 типа Integer. Однако допускается сравнение на равенство двух типов String.

На равенство также можно сравнивать компонентные объекты, если они имеют одинаковый тип. Если быть более точным, они оба должны быть ссылками компонентного объекта, указывающими на компонент того же приложения. В качестве примера приведем следующий метод, находящий все компоненты Reservation, относящиеся к указанному компоненту Customer:

```
public interface ReservationHomeLocal extends EJBLocalObject {
    public Collection findByCustomer(CustomerLocal customer)
        throws FinderException;
    ...
}
```

В соответствующем операторе EJB QL в качестве входного параметра выступает аргумент customer:

```
SELECT OBJECT( r )
FROM Reservation r, IN ( r.customers ) cust
WHERE  cust = ?1
```

Компонентный объект в этом сравнении должен не просто реализовывать интерфейс CustomerLocal, но должен иметь такой же тип, как и компонент Customer, используемый в поле CMR customer компонента Reservation. Другими словами, они должны находиться в одном приложении. Если определено, что компонент является правильным типом, над первичными ключами компонентов выполняется фактическое сравнение. Они считаются равными, если в них содержится одинаковый первичный ключ.

Нельзя проверять на равенство объекты с типом java.util.Data. Для того чтобы сравнить даты, необходимо перевести их в значения long, содержащие количество миллисекунд, а это означает, что дату нужно хранить в поле CMP long, но не java.util.Data. Входное значение или литерал также должны иметь тип long.

Секция WHERE и BETWEEN

Секция BETWEEN – это включающий оператор, определяющий диапазон значений. В данном примере мы используем это для выбора всех судов с весом между 80 000 и 130 000 тоннами:

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage BETWEEN 80000.00 AND 130000.00
```

Секция BETWEEN может применяться только с числовыми примитивами (byte, short, int, long, double, float) и с соответствующими им типами java.lang.Number (Byte, Short, Int и т. д.). Она не может использоваться с типами String, boolean или ссылками на компонентный объект.

Применение логического оператора NOT вместе с BETWEEN приводит к исключению указанного диапазона. Например, следующий оператор EJB QL выбирает все суда, которые весят меньше 80 000 тонн или больше 130 000 тонн, но исключает все значения, попадающие в этот диапазон:

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage NOT BETWEEN 80000.00 AND 130000.00
```

Этот запрос дает такой же результат, как и применение символов сравнения, подобно следующему:

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage < 80000.00 OR s.tonnage > 130000.00
```

Секция WHERE и IN

Условный оператор IN, используемый в секции WHERE, – это не тот же оператор IN, который применяется в секции FROM. В секции WHERE оператор IN проверяет наличие литеральных строковых значений в списке и может использоваться только с операндами, возвращающими строковые значения. Например, в следующем операторе EJB QL оператор IN применяется для выбора всех клиентов, постоянно находящихся в указанном списке штатов:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.state IN ('FL', 'TX', 'MI', 'WI', 'MN')
```

Применение к этому выражению оператора NOT инвертирует данный выбор, исключая всех клиентов, постоянно находящихся в списке штатов:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.city NOT IN ('FL', 'TX', 'MI', 'WI', 'MN')
```

Если проверяемое поле содержит null, значение выражения становится «неизвестным», а это означает, что его значение непредсказуемо.

Секция WHERE и IS NULL

Оператор сравнения IS NULL позволяет проверить, является ли выражение пути нулевым. Например, следующий оператор EJB QL выбирает всех клиентов, не имеющих домашних адресов:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress IS NULL
```

С помощью логического оператора NOT мы можем инвертировать результат этого запроса, выбрав всех клиентов, имеющих домашние адреса:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress IS NOT NULL
```

Когда нулевые поля участвуют в операциях сравнения, таких как IN и BETWEEN, они могут привести к побочным эффектам. В большинстве случаев вычисление нулевого поля в операции сравнения (за исключением IS NULL) производит к неизвестному (UNKNOWN) результату. Такие вычисления ставят под вопрос весь конечный результат EJB QL. Поскольку нельзя предсказать результат такого оператора EJB QL, он является ненадежным. Во избежание этой ситуации можно, например, потребовать, чтобы поля, используемые в выражениях, имели значения. Это требует кропотливого программирования. Можно гарантировать, что поле объектного компонента никогда не будет пустым указателем, но для этого необходимо инициализировать данное поле во время создания объекта. Для примитивных значений это не сложно – они имеют значения по умолчанию и поэтому никогда не бывают нулевыми. Другие поля, такие как одиночные поля CMR и объектные поля CMP, например String, должны инициализироваться в методах ejbCreate() и ejbPostCreate().

Секция WHERE и IS EMPTY

Оператор IS EMPTY позволяет запросу проверить, не пуста ли коллекция отношений. Помните, что основанное на коллекции отношение никогда не будет иметь значение null. Если в коллекции отношений нет никаких элементов, она возвратит пустой объект Collection или Set.

Проверка, выясняющая, не пуста ли коллекция отношений, служит той же цели, что и проверка, выясняющая, не содержит ли одиночное поле CMR или поле CMP значение null: это может применяться для ограничения области запроса и выбранных элементов. Так, следующий запрос выбирает все круизы, на которые не было сделано ни одного заказа:

```
SELECT OBJECT( crs ) FROM Cruise crs
WHERE crs.reservations IS EMPTY
```

Оператор NOT инвертирует результат IS EMPTY. Следующий запрос выбирает все круизы, на которые есть по крайней мере один заказ:

```
SELECT OBJECT( crs ) FROM Cruise crs
WHERE crs.reservations IS NOT EMPTY
```

Нельзя использовать оператор IS EMPTY для коллекции отношений, которым в секции FROM был назначен идентификатор:

```
// неверный запрос
SELECT OBJECT( r )
FROM Reservation r, IN( r.customers ) c
WHERE
r.customers IS NOT EMPTY AND
c.address.city = 'Boston'
```

Этот запрос кажется хорошей страховкой результатов UNKNOWN, но на самом деле это не так. Это неверный оператор EJB QL, потому что оператор IS EMPTY не может применяться для коллекции отношений, идентифицированной в операторе IN секции FROM. Из-за того что это отношение указано в секции IN, в запрос будут включены только те компоненты Reservation, которые имеют непустое поле customers. Все компоненты Reservation, имеющие пустое поле CMR, будут исключены, т. к. его элементы customer не могут быть назначены идентификатору c.

Секция WHERE и MEMBER OF

Оператор MEMBER OF представляет собой мощное средство, позволяющее определить, является ли компонентный объект членом указанной коллекции отношений. Следующий запрос проверяет, является ли определенный Customer (указанный во входном параметре) членом какого-либо отношения Reservation–Customer:

```
SELECT OBJECT( crs )
FROM Cruise crs, IN (crs.reservations) res, Customer cust
WHERE
cust = ?1
  AND
cust MEMBER OF res.customers
```

Применение к MEMBER OF оператора NOT инвертирует результат, выбирая все круизы, для которых указанный клиент не делал заказов:

```
SELECT OBJECT( crs )
FROM Cruise crs, IN (crs.reservations) res, Customer cust
WHERE
cust = ?1
  AND
cust NOT MEMBER OF res.customers
```

Проверка, определяющая, является ли компонентный объект членом пустой коллекции, всегда возвращает `false`.

Секция WHERE и LIKE

Оператор сравнения `LIKE` позволяет запросу выбрать поля CMP типа `String`, соответствующие указанному образцу. Например, следующий оператор EJB QL выбирает всех клиентов с именами, содержащими дефис, такими как «Monson-Haefel» и «Berners-Lee»:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.lastName LIKE '%-%'
```

При задании образца сравнения можно использовать два специальных символа: % (процент), замещающий любую последовательность символов, и _ (символ подчеркивания), замещающий любой одиночный символ. Символы % и _ могут располагаться внутри строкового образца в любом порядке. Если % или _ действительно содержатся в строке, необходимо экранировать их символом \. Логический оператор `NOT` инвертирует выражение таким образом, что совпадающие образцы будут исключены.

Следующие примеры показывают, как секция `LIKE` работает с полями CMP типа `String`:

- `phone.number LIKE '617%'`

  True для «617-322-4151»
  False для «415-222-3523»

- `cabin.name LIKE 'Suite _100'`

  True для «Suite A100»
  False для «Suite A233»

- `phone.number NOT LIKE '608%'`

  True для «415-222-3523»
  False для «608-233-8484»

- `someField.underscored LIKE '\_%'`

  True для «_xyz»
  False для «abs»

- `someField.percentage LIKE '\%%'`

  True для «% XYZ»
  False для «ABC»

Оператор `LIKE` *не может* применяться с входными параметрами. Это – важный момент, являющийся камнем преткновения для многих начинающих разработчиков EJB. Оператор `LIKE` сравнивает поле CMP типа `String` с литералом `String`. В настоящее время он не может использоваться для сравнения входных параметров, потому что входной па-

раметр, по определению, до вызова метода не известен. Во время развертывания известен только тип входного параметра. Как же сравнить с помощью образца сравнения поле CMP, значение которого переменно, с произвольным входным параметром? Это просто невозможно. Образец сравнения составлен из специальных символов сравнения (% и _), а также из обычных символов. Образец сравнения во время развертывания должен быть известен, тогда его можно сопоставить во время выполнения с переменными полями CMP.

Секция WHERE и функциональные выражения

В EJB QL входят шесть функциональных выражений, позволяющих выполнять простые манипуляции со строками и два основных числовых действия. Строковыми функциями являются:

`CONCAT (String1, String2)`

Возвращает строку, представляющую собой результат конкатенации `String1` и `String2`.

`LENGTH(String)`

Возвращает длину строки типа `int`.

`LOCATE(String1, String2 [, start])`

Возвращает позицию типа `int` внутри `String2`, с которой начинается строка `String1`. Необязательный параметр `start` указывает разряд, с которого должен начаться поиск строки `String2`.

`SUBSTRING (String1, start, length)`

Возвращает строку, состоящую из `length` символов строки `String1`, начиная с позиции, заданной `start`.

Параметры `start` и `length` указывают позиции в строке как целочисленные значения. В секции `WHERE` эти выражения позволяют сузить область выбранных элементов. Здесь приведен пример использования функций `LOCATE` и `LENGTH`:

```
SELECT OBJECT( c )
FROM Customer c
WHERE
LENGTH(c.lastName) > 6
  AND
LOCATE( c.lastName, 'Monson') > -1
```

Этот оператор EJB QL выбирает всех клиентов, в фамилиях которых содержится подстрока «Monson», но указывает, что фамилия должна быть длиннее шести символов. Следовательно, «Monson-Haefel» и «Monson-Ares» будут иметь значение `true`, а «Monson» возвратит `false`, т. к. содержит только 6 символов.

Арифметическими функциями EJB QL являются:

`ABS (number)`

Возвращает абсолютную величину числа (`int`, `fload` или `double`).

`SQRT (double)`

Возвращает квадратный корень, результат имеет тип `double`.

Рабочее упражнение 8.2. Сложные операторы EJB QL

Недостатки EJB QL

EJB QL представляет собой мощное новое средство, обещающее увеличить эффективность, гибкость и переносимость объектных компонентов с постоянством, управляемым контейнером, но этот инструмент не свободен от некоторых недостатков и упущений.

Оператор OBJECT ()

Оператор `OBJECT()` является излишним и громоздким и представляет для разработчика компонента минимальную ценность. Производителям EJB очень легко определить, когда возвращаемым значением является абстрактный тип схемы, поэтому реально оператор `OBJECT()` представляет во время трансляции запроса не большую ценность. Кроме того, применять оператор `OBJECT()` небезопасно. Он требуется, когда тип возвращаемого значения представляет собою абстрактный идентификатор схемы, а не тогда, когда выражение пути секции `SELECT` заканчивается полем CMR. Оба возвращают ссылку на компонентный объект, поэтому применение `OBJECT()` в одном сценарии и неприменение в другом производит впечатление нелогичности и запутанности.

На вопросы о том, зачем это понадобилось, в компании Sun ответили, что некоторые производители просили реализовать оператор `OBJECT()`, т. к. он будет включен в будущие версии языка SQL. При создании EJB QL преследовалась цель сделать его похожим на SQL, поскольку язык запросов SQL наиболее знаком разработчикам, но это не значит, что он должен включать функции и операции, не имеющие никакого реального значения в Enterprise JavaBeans.

Отсутствие секции ORDER BY

Начав работу с EJB QL, вы, вероятно, вскоре заметите, что в нем отсутствует важный компонент – секция `ORDER BY`. Возможность запроса упорядоченных списков чрезвычайно важна во всех языках запросов, и основные языки, такие как SQL и объектные языки запросов, поддерживают эту функцию. У секции `ORDER BY` есть два больших преимущества:

- Она предоставляет разработчику компонента понятный механизм для сообщения своих намерений интерпретатору EJB QL. Секция

ORDER BY однозначна, она определенно указывает, как следует упорядочить коллекцию (по каким столбцам, по возрастанию или убыванию и т. д.). Учитывая, что цель EJB QL состоит в том, чтобы ясно описать поведение операций поиска и отбора хорошо переносимым способом, ясно, что отсутствие ORDER BY является значительным упущением.

- В большинстве случаев она позволяет интерпретаторам EJB QL, используемым производителями EJB, выбрать механизм сортировки, оптимизированный под конкретную базу данных. И это более эффективно, чем поручать сортировку контейнеру после извлечения данных.[1] Однако даже если производитель прикладного сервера решает, что сортировку должен выполнять контейнер, секция ORDER BY все еще предоставляет производителю EJB ясное указание на способ сортировки коллекции. А уж как реализовать поддержку секции ORDER BY, можно оставить на усмотрение производителя. В случае применения баз данных и других ресурсов, поддерживающих сортировку, последняя может быть возложена на них. Если ресурсы не поддерживают сортировку, то она может выполняться контейнером. Без секции ORDER BY при развертывании необходимо вручную управлять коллекциями или требовать выполнения этой сортировки от реализации коллекции контейнера. Оба эти варианта в реальных, требовательных к производительности приложениях ненадежны.

Согласно заявлениям корпорации Sun, секция ORDER BY не была включена в эту версию спецификации из-за проблем, связанных с несоответствием в подходах к сортировке языка Java и баз данных. Пример, который они привели, работает со строковыми значениями. Семантика сортировки строк в базе данных может отличаться от принятой в языке Java. Например, Java сортирует типы String в соответствии с последовательностью символов и их регистром (верхний регистр или нижний). Разные базы данных при выполнении сортировки могут учитывать, а могут не учитывать регистр символов, а также могут не учитывать лидирующие и завершающие пробелы. В свете этих возможных различий аргумент Sun кажется вполне разумным, но только с целью ограничения переносимости ORDER BY, а не для полного упразднения ее использования. Разработчики EJB могут прожить с не очень совершенной переносимостью секции ORDER BY, но обойтись совсем без нее они не могут.

Еще один аргумент против секции ORDER BY состоит в том, что она потребует использования в качестве типа возвращаемого значения java.util.List. Хотя тип List предполагается применять для упорядочен-

[1] Предполагалось, что производители EJB смогут реализовать автоматическую сортировку с помощью сортировки полученной коллекции. Это достаточно странное ожидание, т. к. это потребовало бы извлечения всей коллекции, сводя на нет преимущества отложенной загрузки.

ных списков, он также позволяет разработчикам размещать элементы в указанной позиции, что в EJB означало бы определенную позицию в базе данных. Это почти невозможно реализовать, и поэтому кажется, что возражение против секции `ORDER BY` разумно. Однако это слабый аргумент, поскольку не существует ничего, что не разрешало бы использовать в EJB для упорядоченных запросов простой тип `Collection`. Элементы могли бы считаться упорядоченными при условии, что не выполнялось никакой модификации коллекции (т. е. элементы не добавлялись и не удалялись после получения коллекции). Есть и другой вариант. Можно потребовать, чтобы операторы EJB QL, использующие секцию `ORDER BY`, возвращали тип `java.util.Enumeration`. Это кажется весьма разумным, т. к. объектом `Collection`, полученным операцией выборки или поиска, никаким образом нельзя манипулировать.

Хотя Sun не определила поддержку секции `ORDER BY` в EJB QL, ожидается, что некоторые серверы EJB (например, WebLogic от BEA) будут ее поддерживать, так или иначе используя нестандартные расширения. Эта поддержка хотя и приветствуется, но является и достаточно проблематичной из-за того, что нестандартные расширения EJB QL могут привести к непереносимости компонентов.

Недостаточная поддержка даты

EJB QL не предоставляет встроенной поддержки класса `java.util.Date`. Это недопустимо. Класс `java.util.Date` должен поддерживаться в EJB QL как родной тип. Это, например, дало бы возможность сравнивать поля CMP Date, литеральные и входные параметры с помощью операторов сравнения (=, >, >=, <, <=, <>). Также можно было бы ввести общие функции для работы с датой так, чтобы можно было сравнивать даты разных уровней, например дни недели (`DOW()`) или месяца (`MONTH()`) и т. д. Конечно, включение в EJB QL типа `Date` в качестве поддерживаемого типа не тривиально и требует решения вопросов, связанных с интерпретацией дат и их языковой поддержкой, но отказ от рассмотрения даты в качестве поддерживаемого типа является серьезным упущением.

Ограниченная поддержка функциональных выражений

Хотя функциональные выражения, предлагаемые EJB QL, будут полезны разработчикам, также должно быть включено много других функций. Например, в реальных приложениях часто применяется функция `COUNT()`, но в настоящее время она не поддерживается в EJB QL. К другим функциям, которые были бы полезны, относятся `CAST()` (полезная при сравнении разных типов), `MAX()` и `MIN()`, `SUM()` и `UPPER()`. Кроме того, если бы в EJB QL была включена поддержка `java.util.Date`, могли бы быть добавлены и другие функции даты, такие как `DOW()`, `MONTH()` и др.

9

EJB 1.1 CMP

Замечание для тех, кто работает с EJB 2.0

Постоянство, управляемое контейнером, подверглось в EJB 2.0 кардинальному изменению, которое не является обратно совместимым с EJB 1.1. По этой причине производитель EJB 2.0 вынужден поддерживать обе модели постоянства, управляемого контейнером, – и EJB 2.0 и EJB 1.1. Модель EJB 1.1 поддерживается только для обратной совместимости, для того чтобы разработчики могли безболезненно переносить существующие приложения на новую платформу EJB 2.0. Ожидается, что все новые объектные компоненты и новые прикладные программы будут использовать постоянство, управляемое контейнером, версии EJB 2.0, а не EJB 1.1. Хотя постоянство EJB 1.1 рассмотрено в этой книге, следует избегать его применения, если только вам не приходится поддерживать существующие системы EJB 1.1. Постоянство EJB 2.0 рассмотрено в главах 6–8.

Постоянство, управляемое контейнером, в EJB 1.1 ограничено в нескольких отношениях. Например, у компонентов EJB 1.1 CMP могут быть только удаленные интерфейсы, и не может быть ни локальных, ни локальных внутренних интерфейсов. Важнее всего то, что в EJB 1.1 управляемые контейнером объектные компоненты не поддерживают отношения. Существуют также и другие мелкие отличия, делающие EJB 1.1 CMP ограниченным по сравнению с EJB 2.0 CMP. Например, методы `ejbCreate()` и `ejbPostCreate()` в EJB 1.1 не поддерживают суффикс *<ИМЯ-МЕТОДА>*, разрешенный в EJB 2.0, что делает перегрузку ме-

тодов более сложной. EJB 2.0 CMP представляет собою большой шаг вперед по сравнению с CMP 1.1 и должно использоваться везде, где это возможно.

Обзор для тех, кто работает с EJB 1.1

Следующий обзор постоянства, управляемого контейнером, версии EJB 1.1 в значительной степени повторяет сказанное в главе 6, но для тех, кто работает с EJB 1.1 и не читал главу 6, данный обзор важен, т. к. поможет им понять принципы, связанные с объектными компонентами и с постоянством, управляемым контейнером.

В главе 4 мы начали разрабатывать несколько простых компонентов, пропуская массу деталей, относящихся к их созданию. Здесь же мы приведем полный обзор процесса создания объектных компонентов.

Объектные компоненты моделируют прикладные понятия, которые могут быть выражены существительными. Это скорее практическое правило, чем требование, но оно помогает определить, когда прикладное понятие следует реализовывать в виде объектного компонента. Со средней школы нам известно, что существительные – это слова, которые описывают человека, место или предмет. Понятия «человек» и «место» достаточно очевидны: компонент «Человек» может представлять клиента или пассажира, а компонент «Место» – город или пункт назначения. Точно так же объектные компоненты обычно представляют «предметы»: реальные объекты, такие как суда, кредитные карточки и т. д. Объектный компонент может даже представлять нечто довольно абстрактное, вроде заказа билетов. Объектные компоненты описывают и состояние, и поведение реальных объектов и позволяют разработчикам включать в них данные и прикладные правила, связанные с отдельными понятиями. К примеру, компонент Customer содержит данные и прикладные правила, связанные с клиентом. Это делает возможным логичное и безопасное связывание данных с понятием, которым необходимо манипулировать.

В примере с туристической фирмой «Титан» мы можем выделить сотни прикладных понятий, которые обозначаются существительными и поэтому могут быть естественно смоделированы объектными компонентами. Мы уже видели простой компонент Cabin в главе 4, а здесь мы создадим компонент Ship (Корабль). Очевидно, что в примере могли бы использоваться компоненты Customer, Cruise, Reservation и многие другие. Каждое из этих прикладных понятий представляет данные, которые требуется отслеживать и которыми, возможно, необходимо управлять.

Элементы представляют данные в базе данных, поэтому изменения объектного компонента приводят к изменениям в ней. Имеется много причин для того, чтобы предпочесть объектные компоненты непо-

средственному доступу к базе данных. Применение объектных компонентов для хранения данных дает программистам более простой механизм для доступа и изменения данных. Значительно проще, например, поменять имя клиента, вызвав метод `Customer.setName()`, чем выполнять для базы данных команду SQL. Кроме того, хранение данных с помощью объектных компонентов упрощает повторное использование программы. Как только объектный компонент определен, он логичным образом может использоваться во всей системе «Титан». Понятием судна, например, можно оперировать во многих областях системы «Титан», включая регистрацию, планирование расписаний и маркетинг. Компонент Ship предоставляет «Титану» законченный способ доступа к информации о судне. Таким образом гарантируется логичность и простота доступа к информации. Представление данных в виде объектных компонентов может сделать разработку более легкой и более рентабельной.

Во время создания нового компонента в базу данных должна быть вставлена новая запись, а экземпляр компонента необходимо связать с этими данными. В процессе работы с компонентом его состояние меняется, а изменения необходимо синхронизировать с данными в базе данных: записи должны вставляться, модифицироваться и удаляться. Этот процесс координации данных, представляющих экземпляр, и базы данных называется *постоянством* (*persistence*).

Существуют два основных типа объектных компонентов, различающихся тем, как они управляют постоянством. Компоненты с *постоянством, управляемым контейнером*, предполагают, что их постоянство будет автоматически поддерживаться контейнером. Контейнер знает, как поля постоянства экземпляра и его отношения проецируются на базу данных, и автоматически вставляет, модифицирует и удаляет данные, связанные с объектами, из базы данных. Объектные компоненты, *самостоятельно реализующие постоянство*, выполняют всю эту работу вручную: разработчик компонента должен написать код для манипулирования базой данных. Контейнер EJB сообщает экземпляру компонента, когда безопасно вставлять, модифицировать и удалять его данные из базы данных, но не предоставляет никакой другой помощи. Экземпляр компонента делает всю работу по обеспечению своего постоянства самостоятельно. Постоянство, управляемое компонентом, рассматривается в главе 10.

Постоянство, управляемое контейнером

Развертывая объектный компонент EJB 1.1 CMP, необходимо указать поля компонента, которые будут управляться контейнером, и то, как они должны сопоставляться с базой данных. Затем контейнер автоматически генерирует код, необходимый для сохранения состояния экземпляра компонента.

Поля, связываемые с базой данных, называются полями, управляемыми контейнером. EJB 1.1 не поддерживает поля отношений, как это сделано в EJB 2.0. Управляемые контейнером поля могут иметь любой примитивный тип Java или сериализуемого объекта. Большинство компонентов при реализации постоянства с помощью реляционной базы данных используют примитивные типы Java, поскольку связать примитивы Java с типами данных реляционной базы данных значительно проще.

EJB 1.1 позволяет также реализовывать в виде управляемых контейнером полей ссылки на другие компоненты. Для того чтобы это работало, производитель EJB должен обеспечить автоматическое преобразование ссылки (на тип удаленного или внутреннего интерфейса) в нечто, что может быть сохранено в базе данных, а затем преобразовано обратно в удаленную ссылку. Производителям обычно приходится конвертировать удаленные ссылки в первичные ключи, объекты `Handle` и `HomeHandle` или некоторые другие собственные типы указателей, которые могут применяться для хранения ссылки на компонент в базе данных. Контейнер автоматически будет управлять этим преобразованием удаленной ссылки в постоянный указатель и обратно. В EJB 2.0 CMP эта особенность была заменена управляемыми контейнером полями отношений.

Преимущество постоянства, управляемого контейнером, состоит в том, что компонент может быть разработан независимо от того, в какой базе данных будет храниться его состояние. Управляемые контейнером компоненты могут получать доступ к услугам и реляционных и объектно-ориентированных баз данных. Состояние компонента описывается независимо от них, что делает компонент более гибким и пригодным для многократного использования.

Недостаток управляемых контейнером компонентов состоит в том, что они требуют сложных инструментальных средств для описания связи полей компонентов с базой данных. В некоторых случаях потребуется просто сопоставить каждое поле в экземпляре компонента со столбцом базы данных или – в случае сериализации компонента – с файлом. В других случаях это может быть более сложно. Например, состояние некоторых компонентов может быть задано с помощью сложного объединения в реляционной базе данных или отображения на некоторую существующую систему, такую как CICS или IMS.

Здесь мы создадим новый управляемый контейнером объектный компонент Ship (Корабль), который мы изучим подробно. Компонент Ship также фигурирует в главе 7 при обсуждении сложных отношений в EJB 2.0 и в главе 10 при обсуждении постоянства, управляемого компонентом. Одолев эту главу, вы можете сравнить компонент Ship, разработанный здесь, с компонентами, созданными в главах 7 и 10.

Давайте начнем думать о том, что же мы хотим сделать. Огромное количество данных вошло бы в полное описание судна, но для наших

целей мы ограничимся небольшим набором информации. Сейчас мы можем сказать, что судно имеет следующие характеристики (или атрибуты): название, вместимость и тоннаж (т. е. размер). Компонент Ship будет инкапсулировать эти данные, поэтому нам придется создать в нашей базе данных таблицу SHIP для их хранения.

Приведем определение таблицы SHIP с помощью стандартного SQL:

```
CREATE TABLE SHIP (ID INT PRIMARY KEY NOT NULL, NAME CHAR(30), CAPACITY INT,
TONNAGE DECIMAL(8,2))
```

Создание любого компонента мы начинаем с программирования удаленных интерфейсов. Это акцентирует наше внимание на наиболее важном аспекте компонента – его прикладной цели. Определив интерфейсы, мы можем начать работу над действительным определением компонента.

Удаленный интерфейс

Нам понадобится удаленный интерфейс для компонента Ship. В этом интерфейсе определяются прикладные методы, используемые клиентами для взаимодействия с компонентом. Описывая удаленный интерфейс, примем в расчет все отдельные части системы «Титан», которым может потребоваться понятие судна. Здесь приведен удаленный интерфейс ShipRemote компонента Ship:

```
package com.titan.ship;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface ShipRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

Удаленный внутренний интерфейс

Удаленный внутренний интерфейс любого объектного компонента применяется для создания, поиска и удаления объектов из системы EJB. У каждого типа объектных компонентов есть свой собственный внутренний интерфейс. Во внутреннем интерфейсе определяются два основных типа методов: несколько необязательных конструирующих методов и один или несколько поисковых.[1] Конструирующие методы

[1] В главе 15 объясняется, когда следует определять конструирующие методы во внутреннем интерфейсе.

действуют в качестве удаленных конструкторов и определяют, как будут создаваться новые компоненты Ship. (В нашем внутреннем интерфейсе мы ввели только один конструирующий метод.) Поисковый метод предназначен для поиска одного или нескольких компонентов Ship.

Следующий код содержит полное определение интерфейса `ShipHomeRemote`.

```
package com.titan.ship;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Enumeration;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(Integer id, String name,
        int capacity, double tonnage)
        throws RemoteException,CreateException;
    public ShipRemote create(Integer id, String name)
        throws RemoteException,CreateException;
    public ShipRemote findByPrimaryKey(Integer primaryKey)
        throws FinderException, RemoteException;
    public Enumeration findByCapacity(int capacity)
        throws FinderException, RemoteException;
}
```

Enterprise JavaBeans определяет, что конструирующие методы внутреннего интерфейса должны генерировать исключение `javax.ejb.CreateException`. В случае постоянства, управляемого контейнером, контейнеру требуется одно общее исключение для индикации проблем со связью, возникающих во время создания компонента.

Поисковые методы

В EJB 1.1 CMP поддерживаются только поисковые методы, поддержка методов выборки EJB 2.0 не реализована. Кроме того, поисковые методы поддерживаются только в удаленном внутреннем интерфейсе: объектные компоненты EJB 1.1 не поддерживают локальные компонентные интерфейсы.

При использовании постоянства EJB 1.1, управляемого контейнером, реализация поисковых методов автоматически генерируется во время развертывания. Разные производители контейнеров EJB руководствуются различными подходами, определяя, как будут работать поисковые методы. Независимо от реализации во время развертывания компонента нам потребуется выполнить некоторую работу по определению правил для поискового метода. `findByPrimaryKey()` представляет собою стандартный метод, который должны поддерживать все внутренние интерфейсы объектных компонентов. Этот метод ищет компо-

ненты, основываясь на атрибутах первичного ключа. В случае с компонентом Ship первичным ключом является класс `Integer`, соответствующий полю `id` класса `ShipBean`. Для реляционных баз данных атрибуты первичного ключа обычно соответствуют первичному ключу таблицы. К примеру, в классе `ShipBean` атрибут `id` соответствует столбцу первичного ключа `ID` таблицы `SHIP`. В объектно-ориентированной базе данных атрибуты первичного ключа могли бы указывать на какой-то другой уникальный идентификатор.

EJB 1.1, в дополнение к методу `findByPrimaryKey()`, позволяет определять во внутреннем интерфейсе другие поисковые методы. Всем поисковым методам следует давать имена, соответствующие образцу `find<`*`СУФФИКС`*`>()`. Так, если бы мы захотели включить поисковый метод, ищущий суда заданной вместимости, то могли бы назвать его `findByCapacity(int capacity)`. В постоянстве, управляемом контейнером, контейнер должен быть «познакомлен» с каждым поисковым методом, входящим во внутренний интерфейс. Другими словами, управляющий развертыванием должен способом, понятным контейнеру, указать, как этот поисковый метод будет работать. Это делается во время развертывания с помощью инструментальных средств, с использованием уникального для конкретного производителя синтаксиса.

Поисковые методы возвращают либо подходящий для этого компонента тип удаленного интерфейса, либо экземпляр с типом `java.util.Enumeration`, либо `java.util.Collection`. В отличие от EJB 2.0 CMP, EJB 1.1 CMP не поддерживает в качестве типа возвращаемого значения для поисковых методов тип `java.util.Set`.

Указание типа удаленного интерфейса говорит о том, что метод ищет только один компонент. Очевидно, что метод `findByPrimaryKey()` возвращает одиночную удаленную ссылку, поскольку между значением первичного ключа и объектом устанавливается отношение «один-к-одному». Однако метод `findByCapacity(int capacity)` мог бы возвращать несколько удаленных ссылок, по одной для каждого судна, имеющего вместимость, указанную параметром capacity. Для того чтобы иметь возможность возвращать несколько удаленных ссылок, поисковый метод должен возвращать либо тип `Enumeration`, либо `Collection`. Enterprise JavaBeans определяет, что все поисковые методы, применяемые во внутреннем интерфейсе, должны генерировать исключение `javax.ejb.FinderException`. Поисковые методы, возвращающие одиночную удаленную ссылку, генерируют `FinderException`, если происходит прикладная ошибка, и исключение `javax.ejb.ObjectNotFound`, если соответствующий компонент не найден. `ObjectNotFoundException` представляет собой подтип `FinderException` и генерируется *только* поисковыми методами, возвращающими одиночные удаленные ссылки. Поисковые методы, возвращающие типы `Enumeration` или `Collection` (многообъектные поисковые методы), если не найден ни один соответствующий компонент, возвращают пустую коллекцию (а не нулевую

ссылку), а в случае возникновения прикладной ошибки возбуждают исключение `FinderException`.

Способ, которым поисковые методы в постоянстве, управляемом контейнером, будут обращаться к базе данных, не определен в спецификации EJB 1.1 – он специфичен для каждого производителя. Для того чтобы определить, как во время развертывания описываются поисковые методы, обратитесь к документации по вашему серверу EJB. В отличие от CMP в EJB 2.0, здесь нет никакого стандартного языка запросов для задания поведения поисковых методов во время выполнения.

Первичный ключ

Первичный ключ – это объект, уникально идентифицирующий объектный компонент в соответствии с типом компонента, внутренним интерфейсом и контекстом контейнера, в котором он используется. В постоянстве, управляемом контейнером, первичным ключом может являться сериализуемый объект, определенный разработчиком специально для этого компонента, или его определение может быть отложено до начала процесса развертывания. Первичный ключ определяет атрибуты, используемые для поиска отдельного компонента в базе данных. В данном случае нам нужен только один атрибут (`id`), но для первичного ключа возможно наличие нескольких атрибутов, каждый из которых уникально идентифицирует данные компонента. Мы рассмотрим первичные ключи подробно в главе 11, а пока зададим, что компонент Ship использует простой первичный ключ с единственным значением с типом `java.lang.Integer`.

Класс ShipBean

Компонент не считается законченным без реализации его класса. Итак, мы определили удаленные интерфейсы компонента Ship и первичный ключ и готовы определить непосредственно `ShipBean`. `ShipBean` постоянно находится на сервере EJB. Когда клиентское приложение или компонент вызывает прикладной метод удаленного интерфейса компонента Ship, то этот вызов попадет к компонентному объекту, который затем перенаправит его экземпляру `ShipBean`.

При разработке любого компонента мы должны использовать удаленные интерфейсы компонента в качестве руководства. Прикладные методы, определенные в удаленном интерфейсе, должны быть продублированы в классе компонента. Согласно спецификации EJB 1.1, в управляемых контейнером компонентах конструирующие методы внутреннего интерфейса также должны иметь в классе компонента соответствующие им методы. Наконец, методы обратного вызова, содержащиеся в интерфейсе `javax.ejb.EntityBean`, обязательно должны быть реализованы. Далее приведен код класса `ShipBean`:

```
package com.titan.ship;

import javax.ejb.EntityContext;

public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    public int capacity;
    public double tonnage;

    public EntityContext context;

    public Integer ejbCreate(Integer id, String name, int capacity, double
tonnage) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.tonnage = tonnage;
        return null;
    }

    public Integer ejbCreate(Integer id, String name) {
        this.id = id;
        this.name = name;
        capacity = 0;
        tonnage = 0;
        return null;
    }

    public void ejbPostCreate(Integer id, String name, int capacity,
        double tonnage) {
        Integer pk = (Integer)context.getPrimaryKey();
          // Используем первичный ключ
    }

    public void ejbPostCreate(int id, String name) {
        ShipRemote myself = (ShipRemote)context.getEJBObject();
          // Используем ссылку на компонентный объект
    }
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }
    public void unsetEntityContext() {
        context = null;
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}

    public String getName() {
        return name;
    }
    public void setName(String n) {
```

```
        name = n;
    }
    public void setCapacity(int cap) {
        capacity = cap;
    }
    public int getCapacity() {
        return capacity;
    }
    public double getTonnage() {
        return tonnage;
    }
    public void setTonnage(double tons) {
        tonnage = tons;
    }
}
```

В компоненте Ship определено четыре поля постоянства: `id`, `name`, `capacity` и `tonnage`. Эти поля представляют собой состояние постоянства компонента Ship; они определяют уникальный объект Ship в базе данных. Кроме этого, компонент Ship определяет еще одно поле, `context`, содержащее `EntityContext` компонента. Позднее мы поговорим о нем подробно.

Методы set и get – это прикладные методы, определенные нами для компонента Ship; и удаленный интерфейс, и класс компонента должны их поддерживать. Это означает, что сигнатуры этих методов должны быть точно такие же, за исключением `javax.ejb.RemoteException`. Прикладным методам класса компонента не требуется генерировать `RemoteException`. Это имеет смысл, поскольку данные методы фактически не вызываются удаленно – они вызываются компонентным объектом. В случае проблем со связью исключение `RemoteException` будет автоматически сгенерировано контейнером для компонента.

Реализация интерфейса javax.ejb.EntityBean

Будучи объектным компонентом, компонент Ship должен реализовать интерфейс `javax.ejb.EntityBean`. Интерфейс `EntityBean` содержит ряд методов обратного вызова, используемых контейнером для того, чтобы во время выполнения информировать экземпляр компонента о различных событиях:

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
    public abstract void ejbActivate() throws RemoteException;
    public abstract void ejbPassivate() throws RemoteException;
    public abstract void ejbLoad() throws RemoteException;
    public abstract void ejbStore() throws RemoteException;
    public abstract void ejbRemove() throws RemoteException;
    public abstract void setEntityContext(EntityContext ctx)
        throws RemoteException;
    public abstract void unsetEntityContext() throws RemoteException;
}
```

Каждый метод обратного вызова вызывается в определенное время в течение жизненного цикла экземпляра `ShipBean`. В большинстве случаев управляемые контейнером компоненты (такие как компонент Ship) не должны ничего делать, когда происходит вызов метода обратного вызова. Постоянство управляемых контейнером компонентов реализуется автоматически; большинство ресурсов и логики, которые могли бы обрабатываться в этих методах, уже обработаны контейнером.

Текущая версия компонента Ship содержит пустые реализации его методов обратного вызова. Однако следует заметить, что даже управляемый контейнером компонент может при необходимости пользоваться услугами методов обратного вызова; нам они в классе `ShipBean` в настоящее время просто не нужны. Методы обратного вызова подробно рассмотрены в главе 11. Прочитайте эту главу, чтобы больше узнать о методах обратного вызова и о том, когда они вызываются.

Конструирующие методы

Когда вызывается метод `create()` внутреннего интерфейса, внутренний объект перенаправляет его экземпляру компонента таким же образом, как это происходит с прикладными методами удаленного интерфейса. Это значит, что в классе компонента нам нужен метод `ejbCreate()`, который соответствует каждому методу `create()` внутреннего интерфейса.

Метод `ejbCreate()` возвращает нулевое значение типа `Integer` первичного ключа компонента. Значение, возвращаемое методом `ejbCreate()` управляемого контейнером компонента, фактически игнорируется контейнером.

В EJB 1.1 возвращаемое значение метода `ejbCreate()` было изменено с `void`, использованного в EJB 1.0, на тип первичного ключа для того, чтобы облегчить наследование, – это помогает компоненту, самостоятельно реализующему постоянство, расширять компонент, управляемый контейнером. В EJB 1.0 из-за того, что возвращаемое значение имело тип `void`, это не было возможно: Java не даст вам перегрузить методы с разными возвращаемыми значениями. Изменив это определение так, чтобы компонент, управляющий собой самостоятельно, мог расширять управляемый контейнером компонент, спецификация EJB 1.1 дала возможность производителям поддерживать постоянство, управляемое контейнером, путем расширения управляемого контейнером компонента с помощью сгенерированного компонента, реализующего постоянство самостоятельно, – довольно простое решение трудной задачи. Разработчики компонента также могут воспользоваться преимуществами наследования, чтобы изменить существующий CMP-компонент на BMP-компонент, который может потребоваться для решения сложных задач, связанных с постоянством.

Для каждого метода create(), определенного во внутреннем интерфейсе компонента, в классе экземпляра компонента должен существовать соответствующий метод ejbPostCreate(). Другими словами, методы ejbCreate() и ejbPostCreate() встречаются парами с совпадающими сигнатурами. Для каждого метода create(), определенного во внутреннем интерфейсе, должна существовать одна пара.

ejbCreate () и ejbPostCreate ()

В управляемом контейнером компоненте метод ejbCreate() вызывается непосредственно перед записью управляемых контейнером полей компонента в базу данных. Значения, переданные в метод ejbCreate(), должны использоваться для инициализации полей экземпляра компонента. После завершения метода ejbCreate() к базе данных добавляется новая запись, основанная на управляемых контейнером полях.

Разработчик компонента должен обеспечить, чтобы метод ejbCreate() устанавливал поля постоянства, соответствующие полям первичного ключа. Если для управляемого контейнером компонента определен составной первичный ключ, в нем должны быть определены поля, соответствующие одному или нескольким управляемым контейнером полям (постоянства) в классе компонента. Эти поля должны точно соответствовать и по типу, и по имени. Во время выполнения контейнер предполагает, что поля в первичном ключе соответствуют нескольким или всем полям в классе компонента. При создании нового компонента контейнер будет использовать управляемые контейнером поля в классе компонента для автоматического создания и заполнения первичного ключа компонента.

После того как состояние компонента заполнено и установлен его EntityContext, вызывается метод ejbPostCreate(). Этот метод дает компоненту возможность выполнить любую постобработку перед обслуживанием клиентских запросов. Уникальность компонента в течение запроса ejbCreate() еще не реализована, но она становится доступной в методе ejbPostCreate(). Это означает, что компонент может обращаться к его собственному первичному ключу и компонентному объекту, который может быть полезен для инициализации экземпляра компонента перед обслуживанием вызовов прикладных методов. Метод ejbPostCreate() может использоваться для выполнения любой дополнительной инициализации. У каждого метода ejbPostCreate() должны быть такие же параметры, как и у соответствующего ему метода ejbCreate(). Метод ejbPostCreate() возвращает void.

Дополнительную информацию о методах ejbCreate() и ejbPostCreate() и о том, как они связаны с жизненным циклом объектных компонентов, можно найти в главе 11.

Использование ejbLoad() и ejbStore() в управляемых контейнером компонентах

Процесс обеспечения эквивалентности записи базы данных и экземпляра объектного компонента называется *синхронизацией* (*synchronization*). В постоянстве, управляемом контейнером, управляемые контейнером поля компонента синхронизируются с базой данных автоматически. В большинстве случаев нам не потребуются методы `ejbLoad()` и `ejbStore()`, поскольку постоянство в управляемых контейнером компонентах достаточно простое. Эти методы более подробно рассмотрены в главе 11.

Дескриптор развертывания

Закончив с определением компонента Ship, включающим удаленный и внутренний интерфейсы, мы готовы создать дескриптор развертывания. В следующем коде показан XML-дескриптор развертывания компонента. Особенно в нем важен элемент `<cmp-field>`. Эти элементы перечисляют поля, управляемые контейнером. Они имеют то же значение, что и в постоянстве, управляемом контейнером, в EJB 2.0:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <description>
                Этот компонент представляет корабль.
            </description>
            <ejb-name>ShipEJB</ejb-name>
            <home>com.titan.ship.ShipHomeRemote</home>
            <remote>com.titan.ship.ShipRemote</remote>
            <ejb-class>com.titan.ship.ShipBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>capacity</field-name></cmp-field>
            <cmp-field><field-name>tonnage</field-name></cmp-field>
            <primkey-field>id</primkey-field>
        </entity>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>
            Эта роль представляет всех, кому разрешен полный доступ к Ship EJB.
            </description>
```

```
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>ShipEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>ShipEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Элементы `<cmp-field>` перечисляют все управляемые контейнером поля класса объектного компонента. Это те поля, которые будут сохранены в базе данных и которые управляются контейнером во время выполнения.

📖 Рабочее упражнение 9.1. Объектный компонент в CMP 1.1

10

Постоянство, управляемое компонентом

С точки зрения разработчика, постоянство, управляемое компонентом (Bean-Managed Persistence, BMP), требует большего количества усилий, чем постоянство, управляемое контейнером, потому что необходимо явно описать логику постоянства в классе компонента. Чтобы написать код, обрабатывающий постоянство, в классе компонента, требуется знать, какая используется база данных и как с ней соотносятся поля класса компонента.

Если учесть, что постоянство, управляемое контейнером, сберегает много времени, то возникает естественный вопрос – зачем кому-то беспокоиться о постоянстве, управляемом компонентом? Главное преимущество BMP состоит в том, что оно обеспечивает большую гибкость в управлении синхронизацией состояния между экземпляром компонента и базой данных. Компоненты, работающие с данными из нескольких разных баз данных или других ресурсов, таких как уже существующие системы, могут извлечь пользу от применения BMP. По существу, постоянство, управляемое компонентом, является альтернативой постоянству, управляемому контейнером, в тех случаях, когда инструментальные средства контейнера не могут адекватно связать состояние экземпляра компонента с базой данных или другим ресурсом.

Основной недостаток BMP очевиден: создание компонента требует больших затрат. Необходимо понимать структуру базы данных или ресурса и API, применяемый для обращения к ним, а также необходимо разрабатывать логику для создания, модификации и удаления дан-

ных, связанных с объектами. Соответственно, потребуется кропотливая работа при использовании методов обратного вызова EJB (таких как `ejbLoad()` и `ejbStore()`). Кроме того, придется явно создавать поисковые методы, определенные во внутренних интерфейсах компонента.

Методы выборки, используемые в постоянстве, управляемом контейнером, (в EJB 2.0) не поддерживаются в постоянстве, управляемом компонентом.

Еще один недостаток BMP состоит в том, что в случае его применения компонент привязывается к конкретному типу базы данных и ее структуре. Любые изменения в базе данных или структуре данных потребуют изменений в определении экземпляра компонента, и эти изменения могут быть достаточно серьезными. Объект, управляющий постоянством самостоятельно, сильнее зависит от базы данных, чем объект, управляемый контейнером, но он может быть лучше приспособлен к сложному или нетипичному набору данных.

Чтобы вы лучше поняли, как работает BMP, создадим новый компонент Ship, похожий на тот, о котором шла речь в главах 7 и 9. В случае с BMP нам необходимо реализовать методы `ejbCreate()`, `ejbLoad()`, `ejbStore()` и `ejbRemove()` с целью обеспечить согласование состояния компонента с базой данных.

Пользователи EJB 1.1 заметят, что часть информации этой главы взята из главы 9, в которой рассматривалось CMP 1.1. Однако для тех, кто работает с EJB 2.0 и кто, вероятно, пропустил главу 9, большая часть этого материала будет новой. Пользователям EJB 1.1 рекомендуется при встрече со знакомым материалом просматривать его вскользь.

Удаленный интерфейс

Для компонента Ship нам необходимо создать удаленный интерфейс. Этот интерфейс в основном не отличается от любого другого удаленного или локального интерфейса. В нем определяются прикладные методы, применяемые клиентами для взаимодействия с компонентом:

```
package com.titan.ship;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface ShipRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
```

```
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

Мы не будем в этой главе разрабатывать для компонента Ship локальный интерфейс. Однако в EJB 2.0 BMP-объектные компоненты могут иметь и локальный и удаленный компонентные интерфейсы, точно так же, как и CMP-объектные компоненты.

Методы Set и Get

Определение `ShipRemote` использует ряд методов доступа, имена которых начинаются с «set» и «get». Это – не обязательный шаблон, а удобное соглашение, соблюдаемое большинством разработчиков на Java для получения и изменения значений атрибутов или полей объектов. Эти методы часто называются *установщиками* (*setters, mutators*) и *получателями* (*getters, accessors*), а атрибуты, которыми они манипулируют, – *свойствами* (*properties*).[1] Свойства должны быть определены независимо от предполагаемой структуры хранилища данных. Другими словами, следует разрабатывать удаленный интерфейс для моделирования прикладных понятий, а не лежащих в их основе данных. Просто существование свойства `capacity` (вместимость), не означает, что где-то в компоненте или базе данных обязательно должно быть поле `capacity`. Метод `getCapacity()` может вычислять вместимость судна на основе списка кают, найдя модель судна и его конфигурацию, или с помощью какого-то другого алгоритма.

Определение свойств объекта в соответствии с прикладным понятием, а не с лежащими в основе данными не всегда возможно, но необходимо стремиться использовать эту стратегию везде, где это возможно. Причин здесь две. Во-первых, лежащие в основе данные не всегда ясно определяют прикладную цель или понятие, моделируемое объектным компонентом. Удаленные интерфейсы часто используются разработчиками, знакомыми с прикладной задачей, но не знакомыми с конфигурацией базы данных. Для них важно, чтобы объектный компонент адекватно отражал прикладное понятие. Во-вторых, определение свойств компонента независимо от данных позволяет компоненту и данным развиваться самостоятельно. И это достаточно важно – благодаря этому реализация базы данных может быть изменена в любое время. Это также позволяет при необходимости изменить или дополнить поведение объектного компонента. Если определение компонента не зависит от источника данных, то взаимное влияние процессов их разработки минимально.

---

[1] Хотя EJB и отличается от своего GUI-аналога – JavaBeans, – понятия методов доступа и свойств схожи. Об этой идиоме можно больше узнать из книги «Developing JavaBeans» (Разработка JavaBeans), Rob Englander (Роб Ингландер), O'Reilly.

Удаленный внутренний интерфейс

Внутренние интерфейсы объектных компонентов (локальный и удаленный) применяются для создания, поиска и удаления объектов из системы EJB. У каждого типа объектных компонентов есть свой собственный удаленный или локальный внутренний интерфейс. Во внутреннем интерфейсе определяются два основных типа методов: несколько необязательных конструирующих методов и один или несколько поисковых.[1] В данном примере конструирующие методы действуют в качестве удаленных конструкторов и определяют, как будут создаваться новые компоненты Ship. (В нашем внутреннем интерфейсе мы ввели только один конструирующий метод.) Поисковый метод предназначен для поиска одного или нескольких компонентов Ship. Следующий код содержит полное определение интерфейса `ShipHomeRemote`:

```
package com.titan.ship;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(Integer id, String name, int capacity, double
    tonnage)
        throws RemoteException,CreateException;
    public ShipRemote create(Integer id, String name)
        throws RemoteException,CreateException;
    public ShipRemote findByPrimaryKey(Integer primaryKey)
        throws FinderException, RemoteException;

    public Collection findByCapacity(int capacity)
        throws FinderException, RemoteException;
}
```

Enterprise JavaBeans определяет, что конструирующие методы внутреннего интерфейса должны генерировать исключение `javax.ejb.CreateException`. Это дает контейнеру одно общее исключение для индикации проблем со связью, возникающих во время создания компонента.

`RemoteException` возбуждается всеми удаленными интерфейсами и применяется для извещения о сетевых проблемах, возникающих при общении удаленного клиента и контейнерной системы EJB.

---

[1] В главе 15 объясняется, когда следует определять конструирующие методы во внутреннем интерфейсе.

Первичный ключ

В постоянстве, управляемом компонентом, первичный ключ должен быть сериализуемым объектом, определенным разработчиком специально для данного компонента. Первичный ключ определяет атрибуты, которые можно использовать для поиска определенного компонента в базе данных. Для `ShipBean` нам понадобится только один атрибут (`id`), но в других случаях первичный ключ может иметь несколько атрибутов, совместно уникально идентифицирующих данные компонента.

Мы подробно рассмотрим первичные ключи в главе 11, а пока зададим условие, что в компоненте Ship применяется простой первичный ключ с единственным значением типа `java.lang.Integer`. Действительное поле постоянства в классе компонента – это число типа `Integer` с именем `id`.

Класс ShipBean

Класс `ShipBean`, определенный в этой главе для синхронизации состояния компонента с базой данных, использует JDBC. В действительности этот простой объектный компонент мог быть легко развернут в виде компонента CMP. Однако цель этой главы состоит в том, чтобы показать, где в BMP должен применяться код для доступа к ресурсам и как он должен быть реализован. При изучении постоянства, реализуемого компонентом, необходимо сосредоточиться на том, когда и где должно происходить обращение к ресурсам для того, чтобы синхронизировать компонент с базой данных. Тот факт, что мы применяем JDBC и синхронизируем состояние компонента с реляционной базой данных, не особо важен. Компонент мог бы просто быть сохранен в какой-либо существующей системе данных, в приложении ERP или в некотором другом ресурсе, не поддерживаемом в вашей версии CMP, например в LDAP или иерархической базе данных.

Здесь приведено полное определение класса `ShipBean`:

```
package com.titan.ship;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.EntityContext;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import javax.sql.DataSource;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
```

```
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;
import java.util.Collection;
import java.util.Properties;
import java.util.Vector;
import java.util.Collection;

public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    public int capacity;
    public double tonnage;

    public EntityContext context;

    public Integer ejbCreate(Integer id, String name, int capacity, double
    tonnage)
        throws CreateException {
        if ((id.intValue() < 1) || (name == null))
            throw new CreateException("Invalid Parameters");
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.tonnage = tonnage;

        Connection con = null;
        PreparedStatement ps = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement(
                "insert into Ship (id, name, capacity, tonnage) " +
                "values (?,?,?,?)");
            ps.setInt(1, id.intValue());
            ps.setString(2, name);
            ps.setInt(3, capacity);
            ps.setDouble(4, tonnage);
            if (ps.executeUpdate() != 1) {
               throw new CreateException ("Failed to add Ship to database");
            }
            return id;
        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
```

```
    }
    public void ejbPostCreate(Integer id, String name, int capacity,
        double tonnage) {
        // Используем первичный ключ
    }
    public Integer ejbCreate(Integer id, String name) throws CreateException
{
        return ejbCreate(id,name,0,0);
    }
    public void ejbPostCreate(Integer id, String name) {
        // Используем ссылку на компонентный объект
    }
    public Integer ejbFindByPrimaryKey(Integer primaryKey) throws
    FinderException {
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet result = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement("select id from Ship where id = ?");
            ps.setInt(1, primaryKey.intValue());
            result = ps.executeQuery();
            // Существует ли идентификатор судна в базе данных?
            if (!result.next()) {
                throw new ObjectNotFoundException("Cannot find Ship with
                id = "+id);
            }
        } catch (SQLException se) {
            throw new EJBException(se);
        }
        finally {
            try {
                if (result != null) result.close();
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se){
                se.printStackTrace();
            }
        }
        return primaryKey;
    }
    public Collection ejbFindByCapacity(int capacity) throws FinderException
{
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet result = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement("select id from Ship where
            capacity = ?");
            ps.setInt(1,capacity);
```

```
            result = ps.executeQuery();
            Vector keys = new Vector();
            while(result.next()) {
                keys.addElement(result.getObject("id"));
            }
            return keys;

        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
                if (result != null) result.close();
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }
    public void unsetEntityContext() {
        context = null;
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {

        Integer primaryKey = (Integer)context.getPrimaryKey();
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet result = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement("select name, capacity,
                tonnage from Ship where id = ?");
            ps.setInt(1, primaryKey.intValue());
            result = ps.executeQuery();
            if (result.next()){
                id =primaryKey;
                name = result.getString("name");
                capacity = result.getInt("capacity");
                tonnage = result.getDouble("tonnage");
            } else {
                throw new EJBException();
            }
        } catch (SQLException se) {
            throw new EJBException(se);
        }
```

```
        finally {
            try {
                if (result != null) result.close();
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
    public void ejbStore() {
        Connection con = null;
        PreparedStatement ps = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement(
                "update Ship set name = ?, capacity = ?, " +
                "tonnage = ? where id = ?");
            ps.setString(1,name);
            ps.setInt(2,capacity);
            ps.setDouble(3,tonnage);
            ps.setInt(4,id.intValue());
            if (ps.executeUpdate() != 1) {
                throw new EJBException("ejbStore");
            }
        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
    public void ejbRemove() {
        Connection con = null;
        PreparedStatement ps = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement("delete from Ship where id = ?");
            ps.setInt(1, id.intValue());
            if (ps.executeUpdate() != 1) {
                throw new EJBException("ejbRemove");
            }
        }
        catch (SQLException se) {
            throw new EJBException (se);
```

```
            }
            finally {
                try {
                    if (ps != null) ps.close();
                    if (con!= null) con.close();
                } catch(SQLException se) {
                    se.printStackTrace();
                }
            }
        }
        public String getName() {
            return name;
        }
        public void setName(String n) {
            name = n;
        }
        public void setCapacity(int cap) {
            capacity = cap;
        }
        public int getCapacity() {
            return capacity;
        }
        public double getTonnage() {
            return tonnage;
        }
        public void setTonnage(double tons) {
            tonnage = tons;
        }
        private Connection getConnection() throws SQLException {
             // Реализация показана ниже
       }
    }
```

Установка связи с ресурсом

Для того чтобы BMP-объектный компонент мог функционировать, он должен получить доступ к базе данных или ресурсу, в котором он будет хранить свое состояние. Для получения доступа к базе данных компонент обычно должен получить от JNDI ENC фабрику ресурсов (resource factory). JNDI ENC подробно рассмотрен в главе 12, полезно будет привести его краткий обзор и здесь, т. к. фактически он впервые используется в этой книге. Первый шаг при получении доступа к базе данных состоит в запросе соединения от объекта `DataSource`, который мы получаем из окружающего контекста имен JNDI:

```
private Connection getConnection() throws SQLException {
    try {
        Context jndiCntx = new InitialContext();
```

```
            DataSource ds = (DataSource)jndiCntx.lookup("java:comp/env/jdbc/
                                titanDB");
            return ds.getConnection();
        }
        catch (NamingException ne) {
            throw new EJBException(ne);
        }
    }
```

В EJB у каждого компонента есть доступ к своему окружающему контексту имен JNDI (Environment Naming Context, ENC), который является частью соглашения между контейнером и компонентом. Дескриптор развертывания компонента сопоставляет ресурсы, такие как JDBC DataSource, JavaMail и служба сообщений Java, с контекстным именем ENC. Это обеспечивает переносимую модель для доступа к этим типам ресурсов. Здесь приведен фрагмент дескриптора развертывания, описывающий ресурс JDBC:

```
<enterprise-beans>
    <entity>
        <ejb-name>ShipEJB</ejb-name>
        ...
        <resource-ref>
            <description>DataSource for the Titan database</description>
            <res-ref-name>jdbc/titanDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
        ...
    </entity>
    ...
</enterprise-beans>
```

Тег `<resource-ref>` применяется ко всем ресурсам (таким как JDBC, JMS, JavaMail), доступным через ENC. Он описывает JNDI-имя ресурса (`<res-ref-name>`), тип фабрики (`<res-type>`) и определяет, кем будет выполняться аутентификация: непосредственно компонентом или автоматически контейнером (`<res-auth>`). В этом примере мы указали, что JNDI-имя `jdbc/titanDB` ссылается на менеджер ресурсов `javax.sql.DataSource` и что аутентификация базы данных будет автоматически обрабатываться контейнером. JNDI-имя, указанное в теге `<res-ref-name>`, всегда рассматривается относительно имени стандартного контекста JNDI ENC – `java:comp/env`.

При развертывании компонента информация в теге `<resource-ref>` должна быть поставлена в соответствие с реальной базой данных. Это делается способом, специфичным для каждого конкретного производителя, но конечный результат во всех случаях одинаковый. Когда (с использованием JNDI-имени `java:comp/jdbc/titanDB`) запрашивается соединение с базой данных, возвращается объект `DataSource`, соответ-

ствующий базе данных «Титана». За информацией о том, как во время развертывания сопоставить DataSource с базой данных, обратитесь к документации производителя.

Метод getConnection() обеспечивает простой и логичный механизм получения нашим классом ShipBean соединения с базой данных. Итак, у нас есть механизм получения соединения с базами данных, и мы можем использовать его для вставки, модификации, удаления и поиска компонентов Ship в базе данных.

Обработка исключений

Обработка исключений особенно уместна в этом обсуждении по той причине, что в отличие от постоянства, управляемого контейнером, в постоянстве, управляемом компонентом, разработчик ответствен за возбуждение нужных исключений в нужное время. Поэтому мы немного отвлечемся, чтобы рассмотреть различные типы исключений в BMP. Это обсуждение поможет нам, когда мы будем рассматривать детали доступа к базам данных и реализацию методов обратного вызова.

BMP-компоненты генерируют три типа исключений.

Прикладные исключения

К прикладным исключениям относятся стандартные исключения приложения EJB и пользовательские прикладные исключения. К стандартным исключениям приложения EJB относятся CreateException, FinderException, ObjectNotFoundException, DuplicateKey и RemoveException. Они генерируются соответствующими методами для указания на ошибку в прикладной логике. Пользовательские исключения – это исключения, разработанные для указания на специфичные прикладные проблемы. Мы рассмотрим разработку пользовательских исключений в главе 12.

Исключения времени выполнения

Исключения времени выполнения генерируются самой виртуальной машиной и указывают на то, что произошла достаточно серьезная программная ошибка. В частности, к таким исключениям относятся исключения NullPointerException и IndexOutOfBoundsException. Эти исключения автоматически обрабатываются контейнером и не должны обрабатываться внутри метода компонента.

Вы увидите, что все методы обратного вызова (ejbLoad(), ejbStore(), ejbActivate(), ejbPassivate() и ejbRemove()) при возникновении серьезной проблемы генерируют EJBException. Все методы обратного вызова EJB объявляют в своей секции throw исключения EJBException и RemoteException. Все исключения, возбуждаемые методами обратного вызова, должны быть либо EJBException, либо его подклассом. Тип RemoteException включен в сигнатуру метода для обратной совместимости с компонентами EJB 1.0. Его использование не реко-

мендовано еще в EJB 1.1. Исключения RemoteException никогда не должны генерироваться методами обратного вызова компонентов EJB 1.1 и EJB 2.0.

Исключения подсистем

Перехватываемые (checked) исключения, генерируемые другими подсистемами, должны быть вложены внутрь EJBException или прикладного исключения и повторно сгенерированы методом. Так, в предыдущем примере исключение SQLException, сгенерированное JDBC, было перехвачено и возбуждено повторно в виде EJBException. Проверяемые исключения других подсистем, вроде тех, которые генерируются системами JNDI, JavaMail и JMS, должны обрабатываться таким же образом. EJBException представляет собой подтип RuntimeException, поэтому его не нужно объявлять в секции throw метода. Если исключение, генерируемое подсистемой, не достаточно серьезное, можно сгенерировать прикладное исключение, но делать это не рекомендуется, если вы не уверены в причине и результатах исключения в подсистеме. В большинстве случаев предпочтительнее генерировать исключение EJBException.

Исключения воздействуют на транзакции, и значение их при обработке транзакций трудно переоценить. Они более подробно рассматриваются в главе 14.

Метод ejbCreate()

Методы ejbCreate() вызываются контейнером, когда клиент вызывает соответствующий метод create() внутреннего интерфейса компонента. В случае с постоянством, реализуемым компонентом, методы ejbCreate() отвечают за добавление вновь созданных объектов к базе данных. Это означает, что BMP-версия ejbCreate() будет гораздо сложнее аналогичного метода в объектах, управляемых контейнером. При использовании компонентов, управляемых контейнером, от ejbCreate() не требуется ничего, кроме инициализации нескольких полей. Скажем и еще об одном различии между BMP- и CMP-постоянствами. Спецификация EJB определяет, что методы ejbCreate() в постоянстве, управляемом компонентом, должны возвращать первичный ключ только что созданного объекта. В противоположность этому метод ejbCreate() компонентов, управляемых контейнером, должен возвращать null.

Следующий код содержит метод ejbCreate() класса ShipBean. Типом возвращаемого им значения является первичный ключ компонента Ship типа Integer. Для того чтобы вставить новую запись в базу данных, метод использует JDBC API, основываясь на информации, полученной через его параметры:

```
public Integer ejbCreate(Integer id, String name, int capacity, double
tonnage)
```

```
        throws CreateException {
        if ((id.intValue() < 1) || (name == null))
            throw new CreateException("Invalid Parameters");
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.tonnage = tonnage;

        Connection con = null;
        PreparedStatement ps = null;
        try {
            con = this.getConnection();
            ps = con.prepareStatement(
                "insert into Ship (id, name, capacity, tonnage) " +
                "values (?,?,?,?)");
            ps.setInt(1, id.intValue());
            ps.setString(2, name);
            ps.setInt(3, capacity);
            ps.setDouble(4, tonnage);
            if (ps.executeUpdate() != 1) {
                throw new CreateException ("Failed to add Ship to database");
            }
            return id;
        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
```

В начале метода мы проверяем корректность параметров и генерируем `CreateException`, если `id` меньше 1 или `name` равно `null`. Здесь показано, как нужно использовать `CreateException`, чтобы сообщить о прикладной логической ошибке.

Поля экземпляра `ShipBean` инициализируются значениями параметров, переданных методу `ejbCreate()`, путем установки этих полей экземпляра `ShipBean`. Эти значения мы и будем вручную заносить в таблицу `SHIP` нашей базы данных.

Вставку в базу данных мы выполняем посредством метода для SQL-запросов `PreparedStatement` интерфейса JDBC, который помогает увидеть используемые параметры (мы также могли бы обратиться к хранимой процедуре с помощью JDBC `CallableStatement` или простому объекту JDBC `Statement`). Мы вставляем новый компонент в базу данных при

помощи оператора SQL INSERT и значений, переданных методу ejbCreate() через его параметры. Если вставка была выполнена успешно (т. е. не было возбуждено никаких исключений), мы создаем первичный ключ и возвращаем его контейнеру.

Если операция вставки неудачна, мы генерируем новое исключение CreateException, что иллюстрирует применение последнего в менее определенной ситуации. Сбой при вставке записи может считаться прикладной ошибкой или системным отказом. В этой ситуации подсистема JDBC не генерировала исключение, поэтому мы не должны трактовать невозможность вставить запись как отказ подсистемы. Следовательно, мы генерируем вместо EJBException исключение CreateException. Что позволяет приложению восстановиться после ошибки. Этот механизм транзакций будет более подробно рассмотрен в главе 14.

Если операция вставки успешна, метод ejbCreate() должен возвратить контейнеру EJB первичный ключ. В данном случае мы просто возвращаем тот объект Integer, который был передан в метод, но во многих случаях из параметров метода может быть получен новый ключ. Это особенно справедливо, если применяются составные первичные ключи, рассматриваемые в главе 11. За кулисами контейнер использует первичный ключ и экземпляр ShipBean, возвративший его, для того чтобы вернуть клиенту ссылку на новый объект Ship. В общих словах, это означает, что экземпляр ShipBean и первичный ключ связываются с только что созданным компонентным объектом, а клиенту возвращена заглушка компонентного объекта.

Наш внутренний интерфейс требует, чтобы мы предоставили второй метод ejbCreate() с другими параметрами. Мы можем сэкономить время и написать более «пуленепробиваемый» код, сделав так, чтобы второй метод вызывал первый:

```
public Integer ejbCreate(Integer id, String name) throws CreateException {
    return ejbCreate(id,name,0,0);
}
```

Методы ejbLoad() и ejbStore()

В течение всей жизни объекта его данные будут изменяться клиентскими приложениями. В ShipBean мы предоставили методы доступа, предназначенные для изменения названия, вместимости и тоннажа компонента Ship после его создания. Вызов любого из этих методов изменяет состояние экземпляра ShipBean, и эти изменения должны быть отражены в базе данных.

В постоянстве, управляемом контейнером, синхронизация между объектным компонентом и базой данных происходит автоматически, об этом заботится контейнер. В случае с постоянством, управляемым контейнером, за эту синхронизацию отвечает программист: объект-

ный компонент должен читать и записывать информацию непосредственно в базу данных. Контейнер тесно сотрудничает с BMP-объектами, уведомляя их о том, когда необходимо синхронизировать их состояние, с помощью двух методов обратного вызова: `ejbStore()` и `ejbLoad()`.

Метод `ejbStore()` вызывается тогда, когда контейнер «решил», что настало подходящее время для записи данных компонента в базу данных. Контейнер принимает эти решения на основании результатов операций, выполняемых под его управлением, таких как транзакции, параллелизм и управление ресурсами. Реализации от разных производителей могут немного отличаться в том, когда должен вызываться метод `ejbStore()`, но разработчика компонента эти детали не касаются. В большинстве случаев метод `ejbStore()` вызывается после того, как будут вызваны один или несколько прикладных методов или по окончании транзакции.

Далее приведен метод `ejbStore()` для класса `ShipBean`:

```
public void ejbStore() {
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "update Ship set name = ?, capacity = ?, " +
            "tonnage = ? where id = ?");
        ps.setString(1,name);
        ps.setInt(2,capacity);
        ps.setDouble(3,tonnage);
        ps.setInt(4,id.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("ejbStore");
        }
    }
    catch (SQLException se) {
        throw new EJBException (se);
    }
    finally {
        try {
            if (ps != null) ps.close();
            if (con!= null) con.close();
        } catch(SQLException se) {
            se.printStackTrace();
        }
    }
}
```

За исключением случаев, когда выполняется обновление, а не вставка, этот метод похож на метод `ejbCreate()`, рассмотренный выше. Мы используем `PreparedStatement` из JDBC для выполнения команды SQL `UP-`

DATE, а в качестве параметров этого запроса – поля постоянства компонента. Этот метод синхронизирует базу данных с состоянием компонента.

Кроме того, в EJB также введен метод ejbLoad(), синхронизирующий состояние объекта с базой данных. Данный метод обычно вызывается в начале новой транзакции или обращения прикладного метода. Идея состоит в том, чтобы в любой момент обеспечить наличие в компоненте самых свежих данных из базы данных, которые могли быть изменены другими компонентами или приложениями.

Здесь приводится метод ejbLoad() BMP-класса ShipBean:

```
public void ejbLoad() {

    Integer primaryKey = (Integer)context.getPrimaryKey();
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "select name, capacity, tonnage from Ship where id = ?");
        ps.setInt(1, primaryKey.intValue());
        result = ps.executeQuery();
        if (result.next()){
            id = primaryKey;
            name = result.getString("name");
            capacity = result.getInt("capacity");
            tonnage = result.getDouble("tonnage");
        } else {
            throw new EJBException();
        }
    } catch (SQLException se) {
        throw new EJBException(se);
    }
    finally {
        try {
            if (result != null) result.close();
            if (ps != null) ps.close();
            if (con!= null) con.close();
        } catch(SQLException se) {
            se.printStackTrace();
        }
    }
}
```

Для выполнения метода ejbLoad() нам потребуется первичный ключ. А чтобы получить первичный ключ, мы посылаем запрос на получение контекста EntityContext для данного компонента. Обратите внимание, что мы не получаем первичный ключ непосредственно из поля id

класса ShipBean, т. к. не можем быть уверены, что значение этого поля всегда корректно – метод ejbLoad() мог заполнять состояние экземпляра компонента впервые, в этом случае все поля будут установлены в их значения по умолчанию. Такая ситуация могла бы произойти после активации компонента. Мы можем гарантировать только, что EntityContext для ShipBean корректен, поскольку спецификация EJB требует, чтобы ссылка на EntityContext экземпляра компонента перед вызовом метода ejbLoad() была верной.

В этом месте вам, возможно, захочется сразу перейти к разделу, озаглавленному «EntityContext», для того чтобы лучше понять назначение и прикладной смысл EntityContext для объектных компонентов.

Метод ejbRemove()

Кроме обработки своих собственных вставок и обновлений, реализующие постоянство объекты должны обработать и свое собственное удаление. Когда клиентское приложение вызывает метод удаления (remove) внутреннего или компонентного объекта, этот вызов перенаправляется BMP-объекту с помощью вызова ejbRemove(). На разработчике компонента лежит обязанность реализовать метод ejbRemove(), удаляющий данные объекта из базы данных. Ниже приведен метод ejbRemove() для нашего BMP-компонента ShipBean:

```
public void ejbRemove() {
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement("delete from Ship where id = ?");
        ps.setInt(1, id.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("ejbRemove");
        }
    }
    catch (SQLException se) {
        throw new EJBException (se);
    }
    finally {
        try {
            if (ps != null) ps.close();
            if (con!= null) con.close();
        } catch(SQLException se) {
            se.printStackTrace();
        }
    }
}
```

Методы ejbFind()

В постоянстве, управляемом компонентом, поисковые методы удаленного или локального внутренних интерфейсов должны соответствовать методам `ejbFind()` реального класса компонента. Другими словами, для каждого метода с именем `find<`*СУФФИКС*`>()` внутреннего интерфейса в классе компонента должен иметься соответствующий ему метод `ejbFind<`*СУФФИКС*`>()` с теми же параметрами и исключениями. Когда вызывается поисковый метод внутреннего объекта, контейнер перенаправляет этот вызов соответствующему методу `ejbFind()` экземпляра компонента. BMP-объект отвечает за поиск записей, соответствующих этим запросам. В `ShipHomeRemote` определены два поисковых метода:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote findByPrimaryKey(Integer primaryKey)
        throws FinderException, RemoteException;
    public Collection findByCapacity(int capacity)
        throws FinderException, RemoteException;
}
```

Далее приведены сигнатуры соответствующих методов `ejbFind()` класса `ShipBean`:

```
public class ShipBean extends javax.ejb.EntityBean {

    public Integer ejbFindByPrimaryKey(Integer primaryKey)
        throws FinderException {}
    public Collection ejbFindByCapacity(int capacity)
        throws FinderException {}
}
```

Кроме имен есть и другое существенное различие между этими двумя группами методов. Поисковые методы во внутреннем интерфейсе возвращают либо компонентный объект, реализующий удаленный интерфейс компонента (в данном случае – `ShipRemote`), либо коллекцию компонентных объектов в виде `java.util.Enumeration` или `java.util.Collection`. С другой стороны, методы `ejbFind()` класса компонента возвращают либо первичный ключ для соответствующего компонента – в данном случае `Integer`, либо коллекцию первичных ключей. Методы, возвращающие одиночное значение (либо удаленный/локальный интерфейс, либо первичный ключ), применяются тогда, когда необходимо найти одиночную ссылку на компонент. Если же мы ищем группу ссылок (например, все суда определенной вместимости), то должны обратиться к методу, возвращающему либо тип `Enumeration`, либо `Collection`. В любом случае контейнер перехватывает первичные ключи и преобразует их в удаленные клиентские ссылки.

В спецификации EJB 2.0 рекомендуется, чтобы компоненты, самостоятельно реализующие постоянство, вместо типа `Enumeration` использовали тип `Collection`. Эта рекомендация преследовала цель обеспечить большую совместимость BMP-компонентов с EJB 2.0 CMP-компонентами, использующими исключительно тип `Collection`.

Нас не должно удивлять, что возвращаемый тип – является ли он первичным ключом или удаленным (или локальным – в EJB 2.0) интерфейсом – должен соответствовать типу определяемого нами компонента. Например, не следует помещать в компонент Ship поисковые методы для поиска и возвращения объектов Cabin EJB. Если необходимо возвратить коллекцию компонентов разных типов, вызывайте прикладной метод в удаленном интерфейсе, а не поисковый метод одного из внутренних интерфейсов.

В EJB 2.0 контейнер EJB заботится о возвращении клиенту подходящего (локального или удаленного) интерфейса. Например, в компоненте Ship может быть определен и локальный и удаленный внутренний интерфейсы, оба содержащие метод `findByPrimaryKey()`. Когда метод `findByPrimary()` вызывается из локального или удаленного интерфейса, он будет перенаправлен ключевому методу `ejbFindByPrimary()`. После того как метод `ejbFindByPrimaryKey()` будет выполнен и возвратит первичный ключ, контейнер EJB возьмет на себя обязанность возвратить клиенту ссылку `ShipRemote` или `ShipLocal`, в зависимости от того, какой внутренний интерфейс (локальный или удаленный) использовался. В случае многообъектных поисковых методов контейнер EJB обрабатывает их таким же образом, возвращая коллекцию удаленных ссылок для удаленных внутренних интерфейсов или локальных ссылок для локальных внутренних интерфейсов.

Оба поисковых метода, определенные в классе `ShipBean`, генерируют `EJBExeption`, если происходит сбой в запросе, связанный с исключительным условием SQL. Если ни одна запись в базе данных не соответствует параметру `id`, метод `findByPrimaryKey()` генерирует исключение `ObjectNotFoundException`. Это исключение всегда должно генерироваться однообъектными поисковыми методами, если не найден ни один объект.

Метод `findByCapacity()` возвратит пустую коллекцию, если не найдено ни одной записи `SHIP` с соответствующей вместимостью; многообъектные поисковые методы *не* генерируют `ObjectNotFoundExceptions`, даже если не найдено ни одного объекта.

Метод `findByPrimaryKey()` обязателен для всех удаленных и локальных внутренних интерфейсов объекта. Этот метод возвращает тип удаленного интерфейса (в данном случае Ship). В нем объявлен один параметр – первичный ключ для этого типа компонента. В случае локаль-

ных внутренних интерфейсов типом возвращаемого значения любого однообъектного поискового метода всегда является локальный интерфейс компонента. В случае удаленных внутренних интерфейсов тип возвращаемого значения всех однообъектных поисковых методов – удаленный интерфейс. Нельзя развертывать объектный компонент, не содержащий в своих внутренних интерфейсах метод `findByPrimaryKey()`.

Следуя указанным выше правилам, мы можем определить в `ShipBean` два метода `ejbFind()`, соответствующие двум методам `find()`, определенным в `ShipHome`:

```
public Integer ejbFindByPrimaryKey(Integer primaryKey) throws
FinderException, {
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement("select id from Ship where id = ?");
        ps.setInt(1, primaryKey.intValue());
        result = ps.executeQuery();
        // Существует ли ID корабля в базе данных?
        if (!result.next()) {
         throw new ObjectNotFoundException("Cannot find Ship with id = "+id);
        }
    } catch (SQLException se) {
        throw new EJBException(se);
    }
    finally {
        try {
            if (result != null) result.close();
            if (ps != null) ps.close();
            if (con!= null) con.close();
        } catch(SQLException se) {
            se.printStackTrace();
        }
    }
    return primaryKey;
}
public Collection ejbFindByCapacity(int capacity) throws FinderException {
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement("select id from Ship where capacity = ?");
        ps.setInt(1,capacity);
        result = ps.executeQuery();
        Vector keys = new Vector();
        while(result.next()) {
```

```
                keys.addElement(result.getObject("id"));
            }
            return keys;
        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
                if (result != null) result.close();
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
```

Обязательный метод `findByPrimaryKey()` использует первичный ключ для поиска соответствующей записи в базе данных. Как только обнаружено, что данная запись существует, он просто возвращает контейнеру первичный ключ, который затем, когда наступит подходящий момент, использует его для активации нового экземпляра и связывания его с этим первичным ключом. Если с этим первичным ключом не связана ни одна запись, метод генерирует исключение `ObjectNotFoundException`.

Метод `ejbFindByCapacity()` возвращает коллекцию первичных ключей, соответствующих переданным ему критериям. И снова мы создаем подготовленный оператор (prepared statement), который применяем для выполнения нашего запроса SQL. Однако на этот раз мы ожидаем несколько результатов, поэтому используем `java.sql.ResultSet`, чтобы выполнить итерацию по этим результатам и создать вектор первичных ключей для каждого возвращенного `SHIP_ID`.

Поисковые методы не выполняются для тех экземпляров компонентов, которые в настоящее время используются клиентским приложением. Предполагается, что поисковые запросы обслуживают только экземпляры компонентов, не связанные в настоящее время с компонентным объектом (а именно экземпляры в пуле экземпляров). Это означает, что методы `ejbFind()` в экземпляре компонента как-то ограничивают применение `EntityContext`. Методы `getPrimaryKey()` и `getEJBObject()` объекта `EntityContext` сгенерируют исключение, поскольку во время вызова метода `ejbFind()` экземпляр компонента находится в пуле и не связан с первичным ключом или компонентным объектом.

Где же создаются объекты, возвращаемые поисковыми методами? Этот вопрос кажется довольно простым, но ответ на удивление сложен. Вспомните, что поисковые методы не выполняются экземплярами компонентов, которые фактически используются клиентом. Наобо-

рот, контейнер для выполнения этого метода выбирает неактивный экземпляр компонента из пула экземпляров. Контейнер отвечает за создание компонентных объектов и локальных либо удаленных ссылок на первичные ключи, возвращаемые методом `ejbFind()` класса компонента. Как только клиент обращается по этим удаленным ссылкам, экземпляры компонента заменяются подходящими компонентными объектами, наполняются данными и подготавливаются для обслуживания клиентских запросов.

Дескриптор развертывания

Имея готовое определение компонента Ship, включающее удаленный интерфейс, внутренний интерфейс и первичный ключ, мы можем создать дескриптор развертывания. XML-дескрипторы развертывания для BMP-объектных компонентов немного отличаются от дескрипторов, созданных нами в главах 6, 7 и 9 для управляемых контейнером объектных компонентов. В этом дескрипторе развертывания элемент `<persistence-type>` содержит `Bean`, и нет ни одного объявления `<container-managed>` и `<relationship-field>`. Также нам необходимо объявить фабрику ресурсов `DataSource`, которую мы используем для выполнения запросов и обновления базы данных.

Здесь приведен дескриптор развертывания для EJB 2.0:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <description>
                Этот компонент представляет прогулочный корабль.
            </description>
            <ejb-name>ShipEJB</ejb-name>
            <home>com.titan.ship.ShipHomeRemote</home>
            <remote>com.titan.ship.ShipRemote</remote>
            <ejb-class>com.titan.ship.ShipBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <security-identity><use-caller-identity/></security-identity>
            <resource-ref>
               <description>DataSource для базы данных «Титан» </description>
                <res-ref-name>jdbc/titanDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </entity>
    </enterprise-beans>
```

```
    <assembly-descriptor>
        <security-role>
            <description>
              Эта роль представляет всех, кому разрешен полный доступ к Ship EJB.
            </description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>ShipEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>ShipEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Дескрипторы развертывания для EJB 1.1 и EJB 2.0 ничем не отличаются, за исключением двух элементов. Во-первых, элемент `<!DOCTYPE>` ссылается на EJB 1.1 вместо EJB 2.0:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

Во-вторых, элемент `<security-identity>`:

```
<security-identity><use-caller-identity/></security-identity>
```

специфичен для EJB 2.0 и не помещается в дескриптор развертывания EJB 1.1.

📖 Рабочее упражнение 10.1. BMP-объектный компонент

11

Соглашения между объектом и контейнером

Каждый из трех типов объектных компонентов (EJB 2.0 CMP, EJB 1.1 CMP и BMP) программируется по-разному, но их взаимоотношения с контейнерной системой во время выполнения очень похожи. В этой главе рассматриваются отношения между компонентами и их контейнерами. В том числе обсуждаются первичные ключи, методы обратного вызова и жизненный цикл объектных компонентов. Существенные различия между типами компонентов будут выделены особо.

Первичный ключ

Первичный ключ – это объект, уникально идентифицирующий объектный компонент. Первичный ключ может быть любого сериализуемого типа, включая обертки примитивных типов (`Integer`, `Double`, `Long` и т. д.) или пользовательские классы, определенные разработчиком компонента. В компоненте Ship, рассмотренном в главах 7, 9 и 10, первичный ключ имел тип `Integer`. Описания первичных ключей могут быть сделаны разработчиком компонента, а могут быть отложены до начала процесса развертывания. Мы обсудим отложенные первичные ключи (deferred primary keys) немного позже.

Поскольку первичный ключ может использоваться в удаленных вызовах, он должен строго соответствовать ограничениям, наложенным

Java RMI-IIOP, т. е. должен иметь значение одного из допустимых типов Java RMI-IIOP. Эти ограничения рассмотрены в главе 5, но в большинстве случаев просто следует сделать первичный ключ сериализуемым. Кроме того, первичный ключ обязан должным образом реализовывать методы `equals()` и `hashCode()`.

В EJB разрешены два типа первичных ключей: простые и составные. *Простые* (*single-field*) первичные ключи сопоставляются с одиночным полем постоянства, определенным в классе компонента. В компонентах Customer и Ship, например, используется первичный ключ `java.lang.Integer`, соответствующий полю постоянства, управляемого контейнером, с именем `id`. *Составной* (*compound*) первичный ключ – это специально определенный объект, содержащий несколько переменных экземпляра и соответствующий нескольким полям постоянства класса компонента.

Простые первичные ключи

В качестве первичных ключей могут применяться класс String и стандартные обертки для примитивных типов данных (`java.lang.Integer`, `java.lang.Double` и т. д.). Они называются простыми первичными ключами, потому что они неделимы. Они сопоставляются с одним из полей постоянства компонента. Составные первичные ключи соответствуют двум или нескольким полям постоянства.

В компоненте Ship мы определили в качестве первичного ключа тип `Integer`:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public Ship findByPrimaryKey(java.lang.Integer primarykey)
        throws FinderException, RemoteException;
    ...
}
```

В данном случае в классе компонента должно существовать одиночное поле постоянства с тем же самым типом, что и у первичного ключа. В классе `ShipBean` поле CMP `id` имеет тип `java.lang.Integer`, поэтому он хорошо согласуется с типом `Integer` первичного ключа.

В постоянстве, управляемом контейнером, (CMP EJB 2.0) тип первичного ключа должен соответствовать одному из полей CMP компонента. Абстрактные методы доступа для поля `id` в классе `ShipBean` соответствуют этому описанию:

```
public class ShipBean implements javax.ejb.EntityBean {
    public abstract Integer getId();
    public abstract void setId(Integer id);
    ...
}
```

Простой первичный ключ также должен соответствовать полю CMP в постоянстве, управляемом компонентом (рассмотренном в главе 10), и в постоянстве, управляемом контейнером, EJB 1.1 (рассмотренном в главе 9). Для компонента ShipBean, определенного в главах 9 и 10, первичный ключ Integer соответствует полю экземпляра id:

```
public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    ...
}
```

В случае с простыми типами для определения одного из полей CMP компонента в качестве первичного ключа мы указываем сопоставляемое с ним поле постоянства класса компонента с помощью элемента <primkey-field> дескриптора развертывания. Элемент <prim-key-class> задает тип объекта, выступающего в качестве класса первичного ключа. В компоненте Ship при определении поля постоянства id в качестве первичного ключа используются оба этих элемента:

```
<entity>
    <ejb-name>ShipEJB</ejb-name>
    <home>com.titan.ShipHomeRemote</home>
    <remote>com.titan.ShipRemote</remote>
    <ejb-class>com.titan.ShipBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>tonnage</field-name></cmp-field>
    <primkey-field>id</primkey-field>
</entity>
```

Первичными ключами могут быть обертки примитивных типов (Integer, Double, Long и т. д.), но ими не могут быть сами примитивные типы (int, double, long и т. д.), потому что определенная часть семантики интерфейсов EJB запрещает использование примитивов. Например, метод EJBObject.getPrimaryKey() возвращает тип Object, вынуждая таким образом первичные ключи иметь тип Object. Примитивы не могут выступать в качестве первичных ключей еще и потому, что первичные ключи должны вноситься в объекты Collection, работающие только с типами Object. Примитивы не являются типами Object и не имеют методов equals() и hashcode().

Составные первичные ключи

Составной первичный ключ является классом, реализующим интерфейс java.io.Serializable и содержащим одно или несколько откры-

тых полей, имена и типы которых сопоставляются с подмножеством полей постоянства класса компонента. Они определяются разработчиками для конкретных объектных компонентов.

Например, если бы у компонента Ship не было поля `id`, мы могли бы уникально идентифицировать суда по их названиями и регистрационным номерам. (Для этого примера мы добавим к компоненту Ship поле CMP `registration`.) В данном случае поля CMP `name` и `registration` станут нашими полями первичного ключа, которые сопоставляются с соответствующими полями (`NAME` и `REGISTRATION`) в таблице базы данных `SHIP`. Для того чтобы использовать множественные поля в качестве первичного ключа, нам необходимо определить класс первичного ключа.

По соглашению, принятому в этой книге, все составные первичные ключи определяются в виде сериализуемых классов с именами, соответствующими образцу `BeanNamePK`. В этом случае мы можем создать новый класс с именем `ShipPK`, который будет составным первичным ключом для нашего компонента Ship:

```
public class ShipPK implements java.io.Serializable {
    public String name;
    public String registration;

    public ShipPK(){
    }
    public ShipPK(String name, String registration) {
        this.name = name;
        this.registration = registration;
    }
    public String getName() {
        return name;
    }
    public String getRegistration() {
        return registration;
    }
    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof ShipPK))
            return false;

        ShipPK other = (ShipPK)obj;
        if(this.name.equals(other.name) and
            this.registration.equals(other.registration))
            return true;
        else
            return false;

    }
    public int hashCode() {
        return name.hashCode()^registration.hashCode();
    }

    public String toString() {
```

```
            return name+" "+registration;
        }

    }
```

Для того чтобы класс ShipPK мог функционировать как составной первичный ключ, необходимо сделать его поля открытыми. Это позволит контейнерной системе использовать механизм отражения для синхронизации значений в классе первичного ключа с полями постоянства в классе компонента. Мы должны также определить методы equals() и hashCode(), для того чтобы дать возможность контейнерной системе и разработчикам приложений легко манипулировать первичным ключом в составе коллекций.

Важно убедиться, что переменные, объявленные в первичном ключе, имеют соответствующие поля CMP в объектном компоненте с такими же идентификаторами (именами) и типами данных. Это требуется для того, чтобы контейнер с помощью отражения мог сопоставить переменные, объявленные в составном ключе, с нужными полями CMP в классе компонента. В нашем случае переменные экземпляра name и registration, объявленные в классе ShipPK, соответствуют полям CMP name и registration компонента Ship, значит, это соответствие правильно.

Мы также переопределили метод toString(), для того чтобы он возвращал осмысленное значение. Заданная по умолчанию реализация, определенная в Object, возвращает имя класса объекта, добавленного к названию пространства имен.

В классе ShipPK определены два конструктора: конструктор *без аргументов* (*no-argument*) и *перегруженный* (*overloaded*) конструктор, который устанавливает переменные name и registration. Перегруженный конструктор является удобным для разработчиков методом, уменьшающим количество действий, требуемых для создания первичного ключа. Конструктор без аргументов *обязателен* для постоянства, управляемого контейнером. Во время создания нового компонента контейнер с помощью метода Class.newInstance() автоматически создает новый экземпляр первичного ключа и заполняет его содержимым управляемых контейнером полей компонента. Конструктор без аргументов должен существовать, для того чтобы этот процесс работал.

Для того чтобы «подогнать» ShipPK, мы переписываем методы ejbCreate() /ejbPostCreate() так, чтобы они принимали параметры name и registration для установки полей первичного ключа компонента. Здесь показано, как должен использоваться класс первичного ключа ShipPK в классе ShipBean, разработанном нами для EJB 2.0 в главе 7:

```
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class ShipBean implements javax.ejb.EntityBean {

    public ShipPK ejbCreate(String name, String registration) {
```

```
        setName(name);
        setRegistration(registration);
        return null;
    }
    public void ejbPostCreate(String name, String registration) {
    }
    ...
```

В постоянстве EJB 1.1, управляемом контейнером, управляемые контейнером поля устанавливаются непосредственно. Далее показан пример того, как это должно выполняться в компоненте Ship в CMP 1.1:

```
public class ShipBean implements javax.ejb.EntityBean {
    public String name;
    public String registration;

    public ShipPK ejbCreate(String name, String registration) {
        this.name = name;
        this.registration = registration;
        return null;
    }
```

В постоянстве, управляемом компонентом, компонент Ship устанавливает свои поля экземпляра, создает первичный ключ и возвращает его контейнеру:

```
public class ShipBean implements javax.ejb.EntityBean {
    public String name;
    public String registration;

    public ShipPK ejbCreate(String name, String registration){
        this.name = name;
        this.registration = registration;
        ...
        // внесение записей в базу данных
        ...
        return new ShipPK(name, registration);
    }
```

Теперь метод `ejbCreate()` возвращает в качестве первичного ключа объект `ShipPK`. Тип возвращаемого значения метода `ejbCreate()` должен соответствовать типу первичного ключа, если первичный ключ определен, или типу `java.lang.Object`, если не определен.

В постоянстве EJB 2.0, управляемом контейнером, если поля первичного ключа определены, т. е. если они доступны через абстрактные методы доступа, то они должны устанавливаться в методе `ejbCreate()`. Поскольку тип значения, возвращаемого из метода `ejbCreate()`, всегда является типом первичного ключа, возвращаемое значение всегда должно быть равно `null`. Контейнер EJB сам заботится об извлечении подходящего первичного ключа. В постоянстве, управляемом компонентом, за построение первичного ключа и возвращения его контейнеру отвечает класс компонента.

Интерфейс `ShipHomeRemote` должен быть изменен так, чтобы метод `create()` мог использовать параметры `name` и `registration`, а метод `findByPrimaryKey()` — параметр `ShipPK` (EJB требует, чтобы в этом методе использовался тип первичного ключа):

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(String name, String registration)
        throws CreateException, RemoteException;

    public ShipRemote findByPrimaryKey(ShipPK primaryKey)
        throws FinderException, RemoteException;

}
```

Методы `setName()` и `setRegistration()`, модифицирующие поля `name` и `registration` компонента Ship, не следует объявлять ни в удаленных, ни в локальных интерфейсах компонента. Как объяснено в следующем абзаце, первичный ключ объектного компонента не может быть изменен сразу после создания компонента. Однако методы, просто читающие поля первичного ключа (например, `getName()` и `getRegistration()`), могут быть открытыми, т. к. они не меняют значение ключа.

EJB 2.0 указывает, что первичный ключ может быть установлен только один раз либо в методе `ejbCreate()`, либо, если он не определен, автоматически контейнером при создании компонента. После создания компонента поля первичного ключа никогда не должны изменяться ни компонентом, ни одним из его клиентов. Это вполне разумное требование, которое также должно применяться и к EJB 1.1 CMP-, и к BMP-компонентам из-за того, что первичный ключ является уникальным идентификатором компонента. Его изменение могло бы нарушить ссылочную целостность базы данных, возможно, приведя к тому, что два компонента будут соответствовать одному идентификатору, или к разрыву всех отношений с другими компонентами, которые опираются на значения первичного ключа.

Неопределенные первичные ключи

Неопределенные первичные ключи постоянства, управляемого контейнером, были впервые представлены в EJB 1.1. В основном неопределенные первичные ключи позволяют разработчику компонента откладывать объявление первичного ключа до начала процесса развертывания, что дает возможность создавать более переносимые объектные компоненты.

Одна из проблем, связанных с постоянством, управляемым контейнером, в EJB 1.0 состояла в том, что разработчик объектного компонента

должен был определять первичный ключ до развертывания объектного компонента. Это требование вынуждало разработчика делать предположения относительно среды, в которой объектный компонент будет функционировать, что ограничивало переносимость объектного компонента между базами данных. Например, в реляционной базе данных в качестве первичного ключа выступает набор столбцов таблицы, хорошо соотносящихся с полями объектного компонента. Однако в объектной базе данных реализован совершенно другой механизм индексации объектов, с которым первичный ключ не может достаточно хорошо согласовываться. То же самое справедливо и для унаследованных (legacy) систем и систем планирования ресурсов предприятия (Enterprise Resource Planning, ERP).

Неопределенный первичный ключ позволяет выбрать специфичный для конкретной системы ключ во время развертывания. Объектная база данных может генерировать идентификатор объекта, тогда как система ERP может создать какой-то другой первичный ключ. Эти ключи могут быть автоматически сгенерированы базой данных или внутренней системой. Для обеспечения поддержки ключа CMP-компонент может потребовать изменения или расширения средствами развертывания, но для разработчика компонента это уже не важно. Его усилия направлены на реализацию прикладной логики компонента, а индексация возлагается на контейнер.

С целью облегчения применения неопределенных первичных ключей в классе компонента и его интерфейсах для идентификации первичного ключа используется тип `Object`. Компонент Ship, разработанный в главах 7 и 9, мог бы использовать неопределенный первичный ключ. Как показано в следующем фрагменте кода, метод `ejbCreate()` компонента Ship возвращает тип `Object`:

```
public abstract class ShipBean extends javax.ejb.EntityBean {

    public Object ejbCreate(String name, int capacity, double tonnage) {
        ...
        return null;
}
```

Метод `findByPrimaryKey()`, определяемый в локальном и удаленном внутренних интерфейсах, также должен использовать тип `Object`:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote findByPrimaryKey(Object primaryKey)
        throws javax.ejb.FinderException;

}
```

Дескриптор развертывания компонента Ship определяет тип его первичного ключа как `java.lang.Object` и не определяет ни одного элемента `<prim-key-field>`:

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>ShipEJB</ejb-name>
            ...
            <ejb-class>com.titan.ship.ShipBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Object</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>capacity</field-name></cmp-field>
            <cmp-field><field-name>tonnage</field-name></cmp-field>
        </entity>
```

Один из недостатков неопределенного первичного ключа состоит в том, что он требует, чтобы разработчик компонента и разработчик приложения (клиентского кода) работали с типом `java.lang.Object`, а не с конкретным типом первичного ключа, что может привести к определенным ограничениям. Например, невозможно создать неопределенный первичный ключ, чтобы использовать его в поисковом методе, если вы не знаете его тип. Это ограничение может сильно смутить, если необходимо найти объектный компонент по его первичному ключу. Однако объектные компоненты с неопределенными первичными ключами могут быть с легкостью найдены с помощью других методов запроса, не зависящих от значения первичного ключа, поэтому данное ограничение не является очень серьезным препятствием.

В постоянстве, управляемом компонентом, можно объявлять неопределенный первичный ключ, просто создав первичный ключ с типом `java.lang.Object`. Однако это чистая семантика; значение первичного ключа не будет автоматически сгенерировано контейнером, т. к. разработчик компонента имеет над постоянством полный контроль. В данном случае разработчику компонента все еще необходимо использовать действительный первичный ключ, но тип его будет скрыт от клиентов компонента. Этот метод может быть полезен в случаях, когда ожидается, что тип первичного ключа изменится через какое-то время.

Методы обратного вызова

Все объектные компоненты (и управляемые контейнером, и компонентом) должны реализовывать интерфейс `javax.ejb.EntityBean`. Интерфейс `EntityBean` содержит ряд методов обратного вызова, используемых контейнером для оповещения экземпляра компонента о различных событиях времени выполнения:

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
    public abstract void ejbActivate() throws EJBException, RemoteException;
    public abstract void ejbPassivate() throws EJBException, RemoteException;
```

```
    public abstract void ejbLoad() throws EJBException, RemoteException;
    public abstract void ejbStore() throws EJBException, RemoteException;
    public abstract void ejbRemove() throws EJBException, RemoteException;
    public abstract void setEntityContext(EntityContext ctx) throws
EJBException,
        RemoteException;
    public abstract void unsetEntityContext() throws EJBException,
        RemoteException;
}
```

Вызов каждого метода обратного вызова экземпляра объектного компонента происходит в определенное время в течение его жизненного цикла.

Как было описано в главе 10, BMP-компоненты должны реализовывать большую часть этих методов для синхронизации своего состояния с базой данных. Метод `ejbLoad()` сообщает BMP-компоненту, когда можно считывать его состояние из базы данных, `ejbStore()` сообщает ему, когда необходимо записать в базу данных, а `ejbRemove()` сообщает компоненту, когда тот должен удалять себя из базы данных.

Хотя BMP-компонентам доступны все преимущества методов обратного вызова, CMP-объектным компонентам они все могут и не потребоваться. Постоянство CMP-объектных компонентов управляется автоматически, поэтому в большинстве случаев ресурсы и логика, которые могли бы управляться этими методами, уже обработаны контейнером. Однако CMP-объектный компонент может воспользоваться преимуществом методов обратного вызова, если ему нужно выполнить действия, не поддерживаемые контейнером автоматически.

Возможно, вы заметили, что каждый метод в интерфейсе `EntityBean` генерирует и `javax.ejb.EJBException`, и `java.rmi.RemoteException`. В EJB 1.0 это требовалось, чтобы возбуждалось исключение `RemoteException`, если при выполнении компонентом метода обратного вызова имело место системное исключение. Однако в EJB 1.1 применение `RemoteException` в этих методах было отменено в пользу `javax.ejb.EJBException`. EJB 1.1 и EJB 2.0 рекомендуют, чтобы `EJBExecption` генерировались, если при выполнении метода компонент сталкивается с системной ошибкой, такой как `SQLException`. `EJBException` является подклассом `RuntimeException`, поэтому нет необходимости объявлять его в заголовке метода. Поскольку использование `RemoteException` не рекомендуется, не требуется также объявлять его при реализации методов обратного вызова. Фактически рекомендуется, чтобы вы этого не делали.

SetEntityContext() и unsetEntityContext()

Первым после создания экземпляра компонента вызывается метод `setEntityContext()`. Как видно из названия метода, он передает экземпляру компонента ссылку на `javax.ejb.EntityContext`, который является

средством взаимодействия экземпляра компонента с контейнером. Предназначение и функциональность EntityContext подробно рассматриваются далее в этой главе.

Метод setEntityContext() вызывается перед переходом экземпляра компонента в пул экземпляров. В главе 3 мы обсуждали пул экземпляров, поддерживаемый контейнерами EJB, в котором хранятся готовые к использованию объектные компоненты и сеансовые компоненты без состояния. Экземпляры объектных компонентов, находящиеся в пуле, не связаны ни с какими данными в базе данных и их состояние не уникально. Когда клиент запрашивает определенный объект, экземпляр выбирается из пула, заполняется данными из базы данных и назначается для обслуживания клиента. Во время вызова этого метода должны быть получены все неуправляемые ресурсы, необходимые для работы экземпляра. Таким образом гарантируется, что эти ресурсы будут получены только один раз за время жизни экземпляра. Неуправляемый ресурс – это ресурс, который автоматически не поддерживается контейнером (например, ссылки на объекты CORBA). В методе setEntityContext() должны быть получены только те ресурсы, которые не специфичны для уникальности объектного компонента. Другие управляемые ресурсы (например, фабрики Java JMS) и ссылки объектного компонента передаются, как и положено, через JNDI ENC. Ссылки компонента и управляемые ресурсы, получаемые через JNDI ENC, недоступны из setEntityContext(). JNDI ENC рассматривается далее в этой главе.

В конце жизни экземпляра объектного компонента после того, как он навсегда удален из пула экземпляра и до того, как он будет собран сборщиком мусора, вызывается метод unsetEntityContext(), указывая на то, что экземпляр компонента скоро будет удален контейнером из памяти. Это подходящее время для того, чтобы освободить все ресурсы, запрошенные в методе setEntity.Context().

ejbCreate()

В CMP-компоненте метод ejbCreate() вызывается перед тем, как состояние компонента записывается в базу данных. Значения, переданные методу ejbCreate(), должны использоваться для инициализации полей CMP экземпляра компонента. После завершения работы метода ejbCreate() к базе данных будет добавлена новая запись, основанная на полях постоянства.

В постоянстве, управляемом компонентом, метод ejbCreate() вызывается, когда подходит время компоненту внести себя в базу данных. Внутри метода ejbCreate() BMP-компонент должен использовать какой-либо программный интерфейс, чтобы вставить свои данные в базу данных.

Каждый метод ejbCreate() должен иметь параметры, соответствующие методу create() внутреннего интерфейса. Если посмотреть на определение класса ShipBean и сравнить его с внутренним интерфейсом компонента Ship (см. главы 7, 9 и 10), то можно увидеть, что параметры конструирующих методов точно совпадают по типу и порядку следования. Это дает возможность контейнеру перенаправить вызовы всех методов create() внутреннего интерфейса соответствующему методу ejbCreate() экземпляра компонента.

В EJB 2.0 метод ejbCreate() может иметь форму ejbCreate<*SUFFIX*>(), позволяющую достаточно просто перегружать методы с одинаковыми параметрами, но выполняющие разные действия. Например, ejbCreateByName(String name) и ejbCreateByRegistration(String registration) могли бы иметь соответствующие методы create(), определенные в локальном или внутреннем интерфейсе в виде createByName(String name) и createByRegistration(String registration). EJB 1.1 CMP не разрешает использование суффиксов в именах ejbCreate(). Методы ejbCreate() и create() могут отличаться только числом и типом определенных для них параметров.

EntityContext, поддерживаемый экземпляром компонента, до завершения ejbCreate() не обеспечивает объектный компонент достаточной уникальностью. Это означает, что хотя метод ejbCreate() и выполняется, у экземпляра компонента нет доступа ни к его первичному ключу, ни к компонентному объекту. Однако EntityContext дает компоненту информацию, идентифицирующую вызывающий компонент, и доступ к его внутреннему объекту (локальному и удаленному) и его свойствам. Кроме этого для доступа к другим компонентам и менеджерам ресурсов, таким как javax.sql.DataSource, компонент может использовать систему имен контекста JNDI.

Однако разработчик CMP-объектного компонента должен гарантировать, что ejbCreate() устанавливает поля постоянства, соответствующие полям первичного ключа. Во время создания нового CMP-объектного компонента контейнер будет использовать поля CMP в классе компонента для автоматического создания экземпляра и задания значения его первичному ключу. Если первичный ключ объектного компонента не определен, то для его создания контейнер должен работать совместно с базой данных.

Как только состояние компонента будет заполнено, а его EntityContext установлен, вызывается метод ejbPostCreate(). Этот метод дает компоненту возможность выполнить любую постобработку до начала обслуживания клиентских запросов. В CMP-объектных компонентах EJB 2.0 ejbPostCreate() используется для работы с управляемыми контейнером полями отношения. Эти поля CMR не должны измениться методом ejbCreate(). Причина этого ограничения связана со ссылочной целостностью. Первичный ключ для объектного компонента не

может быть доступен до завершения ejbCreate(). Первичный ключ потребуется, если процесс установки отношений использует его в качестве внешнего ключа, поэтому назначение отношений откладывается до завершения ejbPostCreate(), когда первичный ключ станет доступным. Это справедливо и для автоматически генерируемых первичных ключей, которые обычно требуют, чтобы вставка была сделана до того, как может быть сгенерирован первичный ключ. Кроме этого, ссылочная целостность может запретить нулевые внешние ключи в ссылочных таблицах, поэтому вставка должна быть выполнена в первую очередь. В действительности транзакция не завершается до того, как будут выполнены и ejbCreate(), и ejbPostCreate(), поэтому производители могут сами выбрать время для внесения записей в базу данных и связывания отношений.

Идентификатор компонента во время запроса к ejbCreate() недоступен, но он доступен в ejbPostCreate(). Это означает, что из метода ejbPostCreate() компонент может обращаться к своему собственному первичному ключу и компонентному объекту (локальному или удаленному). Это может быть полезно для выполнения постобработки перед началом обслуживания вызовов прикладных методов. В CMP 2.0 ejbPostCreate() может применяться для инициализации полей CMR объектного компонента.

У каждого метода ejbPostCreate() должны быть такие же параметры, как и у соответствующего ему метода ejbCreate(), и такое же имя метода. Например, если класс ShipBean определяет метод ejbCreateByName(String name), он должен также определить соответствующий ему метод ejbPostCreateByName(String name). Метод ejbPostCreate() возвращает void. В EJB 1.1 CMP не допускается наличие суффиксов в названиях конструирующих методов.

Совпадение имен и списков параметров методов ejbCreate() и ejbPostCreate() важно по двум причинам. Во-первых, оно указывает связь методов ejbPostCreate() с методами ejbCreate(). Это гарантирует, что после завершения ejbCreate() контейнер вызовет нужный метод ejbPostCreate(). Во-вторых, возможно, что один из переданных параметров не связан с полем постоянства. В этом случае придется продублировать параметры метода ejbCreate() для того, чтобы сделать эту информацию доступной в методе ejbPostCreate(). Поля отношения являются основной причиной для использования метода ejbPostCreate() в EJB 2.0 CMP из-за ссылочной целостности.

ejbCreate () и ejbPostCreate(): Последовательность событий

Чтобы понять, как экземпляр объектного компонента начинает свою работу, мы должны рассматривать объектный компонент в контексте

его жизненного цикла. На рис. 11.1 отображена последовательность событий некоторого периода жизненного цикла CMP-компонента в том виде, как определяется спецификацией EJB. Каждый производитель EJB должен поддерживать эту последовательность событий.

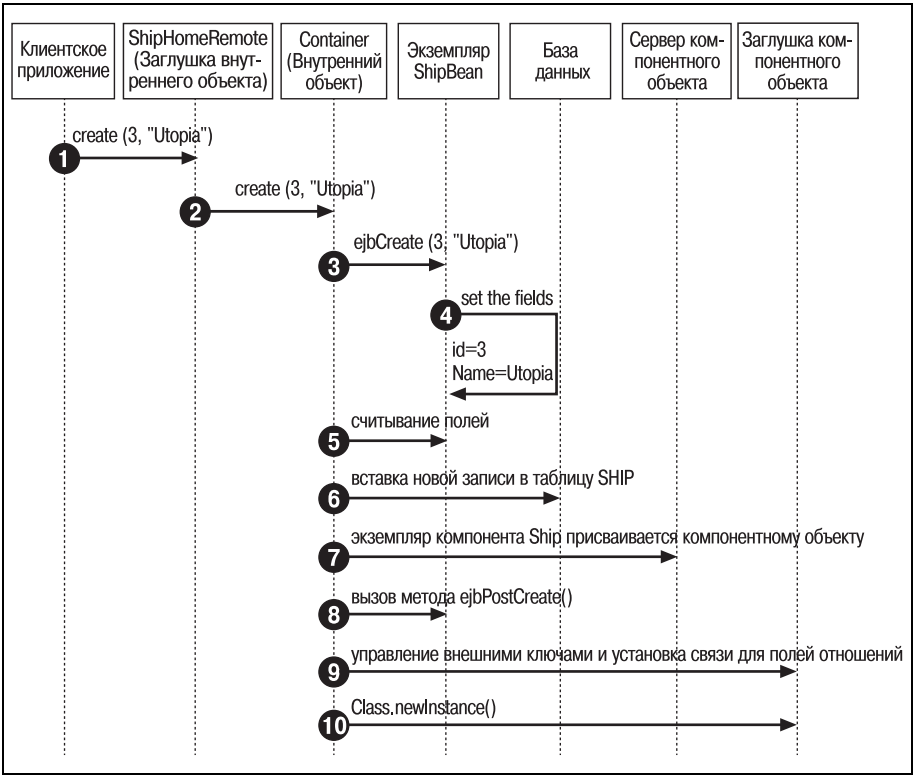

Рис. 11.1. Последовательность создания экземпляра компонента

Процесс начинается, когда клиент вызывает один из методов `create()` внутреннего объекта компонента. Метод `create()` вызывается для заглушки внутреннего объекта (шаг 1), которая через сеть передает этот вызов внутреннему объекту (шаг 2). Внутренний объект «вытаскивает» экземпляр `ShipBean` из пула и вызывает его соответствующий метод `ejbCreate()` (шаг 3).

Методы `create()` и `ejbCreate()` отвечают за инициализацию экземпляра компонента, выполняемую для того, чтобы контейнер мог вставить запись в базу данных. В случае с `ShipBean` минимальная информация, которую требуется включить в систему нового клиента, – уникальный `id` клиента. Это поле CMP инициализируется во время вызова метода `ejbCreate()` (шаг 4).

В постоянстве, управляемом контейнером (EJB 2.0 и 1.1), контейнер использует поля CMP компонента (`id`, `name`, `tonnage`), которые считы-

вает с компонента, для вставки записи в базу данных (шаг 5). Доступными являются только те поля, которые были описаны в дескрипторе развертывания в качестве полей CMP. Прочитав поля CMP экземпляра компонента, контейнер использует их для автоматической вставки в базу данных новой записи (шаг 6).[1] Способ, которым данные записываются в базу данных, определяется тем, каким образом были сопоставлены поля компонента во время развертывания. В нашем примере новая запись вставляется в таблицу `CUSTOMER`.

В постоянстве, управляемом компонентом, для добавления данных компонента в базу данных класс компонента сам читает поля и выполняет вставку в базу данных. Это имело бы место в шагах 5 и 6.

После того как запись вставлена в базу данных, экземпляр компонента может быть присвоен компонентному объекту (шаг 7). После присвоения компонентному объекту становится доступным идентификатор компонента. Это происходит во время вызова метода `ejbPostCreate()` (шаг 8).

В EJB 2.0 в CMP-объектных компонентах метод `ejbPostCreate()` применяется для управления контролируемыми контейнером полями отношений компонентов. Этот процесс может включать установку связи с объектом Cruise в CMR-поле `cruise` компонента Ship или какого-то другого отношения (шаг 9).

Наконец, после окончания работы метода `ejbPostCreate()` компонент готов к обслуживанию клиентских запросов. Создается заглушка компонентного объекта и возвращается клиентскому приложению, которое будет использовать ее для вызова прикладных методов компонента (шаг 10).

Использование ejbLoad() и ejbStore() в постоянстве, управляемом контейнером

Процесс обеспечения эквивалентности записи в базе данных и экземпляра объектного компонента называется *синхронизацией*. В постоянстве, управляемом контейнером, поля CMP компонента автоматически синхронизируются с базой данных. Постоянство в управляемых контейнером компонентах достаточно просто, поэтому в большинстве случаев методы `ejbLoad()` и `ejbStore()` нам не потребуются.

---

[1] В действительности спецификация не требует, чтобы запись вставлялась в базу данных сразу же после вызова метода `ejbCreate()`. В качестве альтернативы вставку записи можно отложить до завершения выполнения метода `ejbPostCreate()` или даже до завершения транзакции.

Однако применение в управляемых контейнером компонентах методов обратного вызова `ejbLoad()` и `ejbStore()` может быть полезно в случаях, когда для синхронизации полей CMP требуются нестандартные действия. Данные, которые требуется хранить в базе данных, могут быть переформатированы или сжаты с целью экономии места; данные, только что полученные из базы данных, могут быть использованы для вычисления производных значений для непостоянных полей.

Представьте себе гипотетический класс компонента, включающий какое-то двоичное значение, которое необходимо сохранить в базе данных. Это двоичное значение может быть очень большим (например в случае сохранения изображения), поэтому может потребоваться его предварительное сжатие. Применение методов `ejbLoad()` и `ejbStore()` в управляемом контейнером компоненте позволяет экземпляру компонента переформатировать данные так, чтобы они соответствовали его состоянию и структуре базы данных. Вот как это могло бы работать:

```
import java.util.zip.Inflater;
import java.util.zip.Deflater;

public abstract class HypotheticalBean implements javax.ejb.EntityBean {
    // переменная экземпляра
    public byte [] inflatedImage;

    // методы поля CMP
    public abstract void setImage(byte [] image);
    public abstract byte [] getImage();

    // прикладные методы. Используются клиентом
    public byte [] getImageFile() {
        if(inflatedImage == null) {
            Inflater unzipper = new Inflater();
            byte [] temp = getImage();
            unzipper.setInput(temp);
            unzipper.inflate(inflatedImage);
        }
        return inflatedImage;
    }
    public void setImageFile(byte [] image) {
        inflatedImage = image;
    }

    // методы обратного вызова
    public void ejbLoad() {
        inflatedImage = null;
    }
    public void ejbStore() {
        if(inflatedImage != null) {
            Deflater zipper = new Deflater();
            zipper.setInput(inflatedImage);
```

```
                byte [] temp = new byte[inflatedImage.length];
                int size = zipper.deflate(temp);
                byte [] temp2 = new byte[size];
                System.arraycopy(temp, 0, temp2, 0, size);
                setImage(temp2);
            }
        }
    }
```

Непосредственно перед тем, как контейнер начнет синхронизировать состояние объектного компонента с базой данных, он вызывает метод `ejbStore()`. Перед записью файла изображения в базу данных (если оно было изменено) этот метод сжимает его с помощью пакета `java.util.zip`.

Сразу же после того, как контейнер обновит поля `HypotheticalBean` новыми данными из базы данных, он вызывает `ejbLoad()`, который повторно инициализирует переменную `inflatedImage` экземпляра, присваивая ей значение `null`. Декомпрессия выполняется медленно, поэтому она происходит только тогда, когда это необходимо. Сжатие выполняется в методе `ejbStore()`, только если к изображению выполнялся доступ; иначе поле изображения не изменяется.

Использование ejbLoad() и ejbStore() в постоянстве, управляемом компонентом

В постоянстве, управляемом компонентом, методы `ejbLoad()` и `ejbStore()` вызываются контейнером, когда наступает время читать или записывать в базу данных. Метод `ejbLoad()` вызывается после начала транзакции, но перед тем как объектный компонент начнет обслуживать запросы методов. Метод `ejbStore()` обычно вызывается после прикладного метода, но он должен быть вызван до завершения транзакции.

В то время как объектный компонент отвечает за чтение и запись своего состояния в базе данных, контейнер отвечает за управление областью транзакции. Это означает, что контейнер предоставляет разработчику объектного компонента все необходимые ресурсы, избавляя его от забот, связанных с выполнением фиксирующих операций (committing operation) в программном интерфейсе для доступа к базе данных. Контейнер будет заботиться о фиксации транзакции и сохранении изменений в нужное время.

Если BMP-объектный компонент использует ресурс, не поддерживаемый контейнерной системой, то в компоненте должно быть реализовано управление областью транзакции вручную, с помощью действий, специфичных для конкретного API. Детальные примеры такого использования методов `ejbLoad()` и `ejbStore()` в BMP приведены в главе 10.

ejbPassivate() и ejbActivate()

Метод `ejbPassivate()` уведомляет разработчика компонента о том, что экземпляр компонента будет помещен в пул, или, иначе говоря, отсоединен от уникальности компонента. Это дает разработчику компонента возможность выполнить последнюю приборку прежде, чем компонент будет помещен в пул, где он затем будет повторно использован каким-то другим компонентным объектом. В рабочих реализациях метод `ejbPassivate()` применяется достаточно редко из-за того, что большинство ресурсов бывают получены с помощью JNDI ENC и автоматически поддерживаются контейнером.

Метод `ejbActivate()` уведомляет разработчика компонента о том, что экземпляр объектного компонента только что был извлечен из пула, связан с компонентным объектом и получил свой идентификатор. Это дает разработчику компонента возможность подготовить объектный компонент к работе, установив, например, связь с некоторым ресурсом.

Как и в случае с методом `ejbPassivate()`, достаточно непросто понять, зачем этот метод нужен на практике. Лучше подключать ресурсы отложенно (т. е. при необходимости). Метод `ejbActivate()` предназначен для проведения некоторой неотложной подготовки, но фактически он применяется редко.

Даже в контейнерах EJB, которые не используют пул экземпляров объектных компонентов, потребность в `ejbActivate()` и `ejbPassivate()` сомнительна. Возможно, что контейнер EJB может решить выгружать экземпляры из памяти в промежутках между обращениями клиентов и создавать новый экземпляр для каждой новой транзакции. Хотя может показаться, что такая стратегия только ухудшит производительность, она достаточно разумна при условии, что виртуальная машина Java контейнерной системы имеет чрезвычайно эффективный механизм «сборки мусора» и стратегию распределения памяти. Hotspot является примером JVM, добившейся в этом направлении больших успехов. Однако даже в этом случае `ejbActivate()` и `ejbPassivate()` не представляют большой ценности, т. к. методы `setEntityContext()` и `unsetEntityContext()` могут выполнять то же самое.

Одна из немногих практических причин для использования `ejbActivate()` состоит в повторной инициализации непостоянных полей экземпляра класса компонента, которые могут «испортиться» за то время, пока экземпляр обслуживал другого клиента.

Независимо от их общей ценности, эти методы обратного вызова предоставлены в распоряжение программиста на тот случай, если они ему понадобятся. В большинстве случаев стоит предпочесть `setEntityCon-`

text() и unsetEntityContext(), т. к. эти методы выполнятся только один раз в жизненном цикле экземпляра компонента.

EjbRemove()

Компонентные интерфейсы (удаленный, локальный, удаленный внутренний и локальный внутренний) определяют удаляющие методы, используемые для удаления объекта из системы. Когда клиент вызывает один из удаляющих методов, как показано в следующем коде, контейнер должен удалить данные объекта из базы данных:

```
CustomerHomeRemote customerHome;
CustomerRemote customer;

customer.remove();
// или
customerHome.remove(customer.getPrimaryKey());
```

Данные удаляются из базы данных вместе со всеми полями CMP. Например, когда мы вызываем удаляющий метод на компоненте Customer, удаляется соответствующая ему запись в таблице CUSTOMER.

В CMP 2.0 удаляющий метод также разрывает связь между записью CUSTOMER и записью ADDRESS. Однако запись ADDRESS, связанная с записью CUSTOMER, автоматически удалена не будет. Данные компонента Address будут удалены вместе с данными компонента Customer, только если было задано каскадное удаление. Каскадное удаление должно быть объявлено непосредственно в XML-дескрипторе развертывания, как это объяснялось в главе 7.

Метод ejbRemove() в постоянстве, управляемом контейнером, сообщает объектному компоненту, что он и его данные скоро будут удалены. Это уведомление посылается после того, как клиент вызывает один из удаляющих методов, определенных в компонентном интерфейсе, но до того как контейнер действительно удалит данные. Это дает разработчику компонента возможность выполнить последнюю «приборку» перед удалением объекта. Любые действия по очистке, которые первоначально могли бы быть сделаны в методе ejbPassivate(), должны также быть выполнены в методе ejbRemove(), т. к. компонент будет помещен в пул после завершения метода ejbRemove() без необходимости вызывать его метод ejbPassivate().

В постоянстве, управляемом компонентом, разработчик компонента отвечает за реализацию действий по удалению данных объектного компонента из базы данных.

EJB 2.0: ejbHome()

В EJB 2.0 CMP- и BMP-объектные компоненты могут объявлять *внутренние методы* (*home methods*), выполняющие действия, связанные

с компонентом, но не относящиеся к конкретному экземпляру объектного компонента. Внутренний метод должен иметь соответствующую реализацию в классе компонента с сигнатурой ejbHome<*METHODNAME*>().

Например, компонент Cruise может определить внутренний метод, вычисляющий совокупный доход, поступивший от заказов на отдельный круиз:

```
public interface CruiseHomeLocal extends javax.ejb.EJBLocalHome {

    public CruiseLocal create(String name, ShipLocal ship);
    public void setName(String name);
    public String getName();
    public void setShip(ShipLocal ship);
    public ShipLocal getShip();

    public double totalReservationRevenue(CruiseLocal cruise);

}
```

Все методы во внутренних интерфейсах должны иметь соответствующие методы ejbHome<*METHODNAME*>() в классе компонента. Например, класс CruiseBean должен содержать метод ejbHomeTotalReservationRevenue(), как показано в следующем фрагменте кода:

```
public abstract class CruiseBean implements javax.ejb.EntityBean {
    public Integer ejbCreate(String name, ShipLocal ship) {
        setName(name);
    }
    ...
    public double ejbHomeTotalReservationRevenue(CruiseLocal cruise) {

        Set reservations = ejbSelectReservations(cruise);
        Iterator enum = set.iterator();
        double total = 0;
        while(enum.hasNext()) {
            ReservationLocal res = (ReservationLocal)enum.next();
            Total += res.getAmount();
        }
        return total;

    }

   public abstract ejbSelectReservations(CruiseLocal cruise);
   ...
}
```

Методы ejbHome() выполняются, когда компонент находится в пуле и не представляет конкретный объект, т. е. не обладает уникальностью. Вот почему требуется передать методу ejbHomeTotalReservationRevenue() ссылку на объект компонента CruiseLocal. Это имеет смысл, если вы понимаете, что вызывающий объект обращается к внутреннему мето-

ду внутреннего объекта объектного компонента, а не прямо к ссылке объектного компонента. Внутренний объект (локальный или удаленный) не относится к конкретному экземпляру объекта.

Разработчик компонента может реализовать внутренние методы в обеих реализациях постоянства для EJB 2.0 – и в управляемом компонентом, и в управляемом контейнером, – в основном, посредством методов выборки, тогда как в BMP-реализации для выполнения запроса данных и их изменения часто применяются поисковые методы и прямой доступ к базе данных.

EntityContext

Метод `setEntityContext()` первый вызывается контейнером после создания экземпляра компонента. Этот метод передает экземпляру компонента ссылку на его `javax.ejb.EntityContext`, который в действительности является интерфейсом между экземпляром и контейнером.

Метод `setEntityContext()` должен быть реализован разработчиком объектного компонента так, чтобы он помещал ссылку `EntityContext` в поле экземпляра компонента, где она будет храниться на протяжении всего времени жизни экземпляра. Определение `EntityContext` в EJB 2.0 следующее:

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext {
    public EJBLocalObject getEJBLocalObject() throws IllegalStateException
    public abstract EJBObject getEJBObject() throws IllegalStateException;
    public abstract Object getPrimaryKey() throws IllegalStateException;
}
```

`EJBLocalObject` появился в EJB 2.0 и не поддерживается в EJB 1.1. Определение `EntityContext` в EJB 1.1 следующее:

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject() throws IllegalStateException;
    public abstract Object getPrimaryKey() throws IllegalStateException;
}
```

Поскольку экземпляр компонента передается от одного компонентного объекта другому, информация, получаемая через `EntityContext`, меняется для того, чтобы отражать тот компонентный объект, которому присваивается экземпляр. Данное изменение возможно из-за того, что `EntityContext` – это интерфейс, а не статическое определение класса, и это означает, что контейнер может реализовать `EntityContext` с помощью контролируемого им конкретного класса. Экземпляр объектного компонента передается от одного компонентного объекта другому, поэтому некоторая часть информации, ставшая доступной через `EntityContext`, также будет меняться.

Метод `getEJBObject()` возвращает удаленную ссылку на экземпляр компонентного объекта. С другой стороны, метод `getEJBLocalObject()` (EJB 2.0) возвращает локальную ссылку на экземпляр компонентного объекта.

Сеансовые компоненты также определяют методы `getEJBObject()` и `getEJBLocalObject()` (EJB 2.0) в интерфейсе `SessionContext`; их поведение точно такое же.

Компонентные объекты, получаемые через `EntityContext`, являются в случае удаленной ссылки такими же ссылками, которые могут использоваться клиентским приложением, или в случае локальной ссылки – еще одним совмещенным компонентом. Эти методы позволяют экземпляру компонента получить свою собственную ссылку на компонентный объект, которую он затем может передавать другим компонентам. Это иллюстрируется следующим примером:

```
public class A_Bean extends EntityBean {
    public EntityContext context;
    public void someMethod() {
        B_Bean  b = ... // получаем удаленную ссылку на компонент B
        EJBObject obj = context.getEJBObject();
        A_Bean mySelf =
(A_Bean)PortableRemoteObject.narrow(obj,A_Bean.class);
        b.aMethod( mySelf );
    }
    ...
}
```

Экземпляр компонента не может передавать другому компоненту ссылку `this`. Вместо этого он передает удаленную или локальную ссылку на свой компонентный объект, которую экземпляр компонента получает через его `EntityContext`.

В EJB 2.0 способность компонента получать компонентную ссылку на самого себя также бывает полезна и при установлении отношений с другими компонентами в постоянстве, управляемом контейнером. Например, компонент Customer может реализовать прикладной метод, который позволяет ему связать себя с компонентом Reservation:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    public EntityContext context;

    public void assignToReservation(ReservationLocal reservation) {
        EJBLocalObject localRef = context.getEJBLocalObject();
        Collection customers = reservation.getCustomers();
        customers.add(localRef);
```

```
        }
        ...
    }
```

Метод `getPrimaryKey()` позволяет экземпляру компонента получить копию первичного ключа, с которым он в настоящее время связан. Применение этого метода вне методов `ejbLoad()` и `ejbStore()` BMP-объектных компонентов достаточно редко, но `EntityContext` делает первичный ключ доступным для некоторых необычных ситуаций, когда он бывает необходим.

Контекст, в котором выполняется экземпляр компонента, изменяется, поэтому некоторая часть информации, ставшая доступной через ссылку `EntityContext`, будет изменена контейнером. Вот почему методы в `EntityContext` генерируют `java.lang.IllegalStateException`. `EntityContext` всегда доступен для экземпляра компонента, но экземпляр не всегда прикреплен к компонентному объекту. Когда компонент находится среди компонентных объектов (т. е. в пуле), у него нет ни компонентного, ни первичного ключа, которые можно было бы вернуть. Если методы `getEJBObject()`, `getEJBLocalObject()` или `getPrimaryKey()` вызываются в то время, когда компонент находится в пуле, они возбудят исключение `IllegalStateException`. В приложении B приведены таблицы операций, разрешенных для каждого типа компонента, описывающие, в какое время какие методы `EJBContext` могут быть вызваны.

EJBContext

`EntityContext` расширяет класс `javax.ejb.EJBContext`, который, в свою очередь, является базовым классом для `SessionContext`, используемого сеансовыми компонентами. В `EJBContext` определено несколько методов, предоставляющих компоненту полезную информацию во время выполнения. Здесь приведено определение интерфейса `EJBContext`:

```
package javax.ejb;
public interface EJBContext {

    // внутренние методы компонента
    public EJBHome getEJBHome();
    // только для EJB 2.0
    public EJBLocalHome getEJBLocalHome();

    // методы безопасности
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String roleName);

    // методы транзакций
    public javax.transaction.UserTransaction getUserTransaction()
        throws java.lang.IllegalStateException;
    public boolean getRollbackOnly()
        throws java.lang.IllegalStateException;
```

```
    public void setRollbackOnly()
        throws java.lang.IllegalStateException;

    // отмененные методы
    public java.security.Identity getCallerIdentity();
    public boolean isCallerInRole(java.security.Identity role);
    public java.util.Properties getEnvironment();

}
```

Методы `getEJBHome()` и `getEJBLocalHome()` **(EJB 2.0) возвращают ссылку на внутренний объект компонента. Это полезно, если компоненту необходимо создать или найти объектные компоненты одного с ним типа. Доступ к внутреннему объекту оказывается более полезным в BMP-объектных компонентах и CMP 1.1-компонентах, чем в CMP 2.0-компонентах, имеющих поля CMR и методы выборки.**

Как пример, если весь персонал системы «Титан» (включая менеджеров) представить CMP 1.1-компонентами Employee, то менеджер, которому нужен доступ к своим подчиненным, сможет использовать для получения компонентов, представляющих требуемых рабочих, метод `getEJBHome()`**:**

```
public class EmployeeBean implements EntityBean {
    int id;
    String firstName;
    ...
    public Enumeration getSubordinates() {
        Object ref = ejbContext.getEJBHome();
        EmployeeHome home = (EmployeeHome)
            PortableRemoteObject.narrow(ref, EmployeeHome.class);
        Integer primKey = (Integer)context.getPrimaryKey();
        Enumeration subordinates = home.findByManagerID(primKey);
        return subordinates;
    }
    ...
}
```

Метод `getCallerPrincipal()` **вызывается для получения объекта** `Principal`**, представляющего клиента, который в настоящее время обращается к компоненту. Объект** `Principal` **может использоваться, например, компонентом Ship, чтобы проследить идентификаторы клиентов, вносящих изменения:**

```
public class ShipBean implements EntityBean {
    String lastModifiedBy;
    EntityContext context;
    ...
    public void setTonnage(double tons) {
        tonnage = tons;
        Principal principal = context.getCallerPrincipal();
```

```
            String modifiedBy = principal.getName();
        }
        ...
    }
```

Метод `isCallerInRole()` сообщает, является ли клиент, обращающийся к компоненту, членом определенной роли, идентифицируемой по ее имени. Этот метод полезен, когда требуется больший контроль доступа, чем тот, который может быть обеспечен на базе методов. Например, в банковской системе мы могли бы разрешить роли Teller (кассир) делать большую часть выплат, но выплаты свыше $10 000 разрешается делать только роли Manager (управляющий). Этот вид разделения управления доступом не может быть реализован посредством атрибутов безопасности компонентов, т. к. он связан с задачей прикладной логики. Поэтому мы можем вызвать метод `isCallerInRole()` для усиления автоматического управления доступом, обеспечиваемого EJB. Для начала предположим, что все менеджеры являются также и кассирами. Прикладная логика в методе `withdraw()` использует `isCallerInRole()`, чтобы убедиться, что только роль Manager может выдавать суммы, превышающие $10 000:

```
public class AccountBean implements EntityBean {
    int id;
    double balance;
    EntityContext context;

    public void withdraw(Double withdraw) throws AccessDeniedException {

        if (withdraw.doubleValue() > 10000) {
            boolean isManager = context.isCallerInRole("Manager");
            if (!isManager) {
                // более 10 000 могут выдавать только менеджеры
                throw new AccessDeniedException();
            }
        }
        balance = balance - withdraw.doubleValue();

    }
    ...
}
```

`EJBContext` содержит несколько методов, которые использовались в EJB 1.0, но стали нерекомендуемыми в EJB 1.1 и были исключены в EJB 2.0. Поддержка для этих отмененных методов не обязательна для контейнеров EJB 1.1, которые могут выполнять компоненты EJB 1.0. Контейнеры EJB, не поддерживающие отмененные методы безопасности, будут генерировать исключение `RuntimeException`. Отмененные методы безопасности основаны на использованном в EJB 1.0 объекте `Identity` вместо объекта `Principal`. Семантика отмененных методов в ос-

новном такая же, но из-за того что `Identity` – это абстрактный класс, его оказалось слишком трудно использовать.

Метод `getEnvironment()` был заменен контекстом имен окружения JNDI, обсуждаемым далее в этой книге. Поддержка для отмененного метода `getEnvironment()` в EJB 1.1 подробно рассмотрена в главе 12. Методы транзакций – `getUserTransaction()`, `setRollbackOnly()` и `getRollbackOnly()` – подробно описаны в главе 14.

Материал по `EJBContext`, рассмотренный в этом разделе, одинаково применим и к сеансовым, и к управляемым сообщениями компонентам. Однако существует несколько исключений, и эти различия рассматриваются в главах 12 и 13.

JNDI ENC

Начиная с EJB 1.1 соглашения между контейнером и компонентами для объектных компонентов и сеансовых компонентов с состоянием были расширены за границы `EJBContext` с помощью интерфейса идентификации и маршрутизации Java (Java Naming and Directory Interface, JNDI). В JNDI было добавлено специальное пространство имен, называемое *контекстом имен окружения* (*Environment Naming Context, ENC*), для того чтобы разрешить любому компоненту обращаться к элементам среды, другим компонентам и ресурсам (таким как объекты `DataSource` в JDBC), необходимым данному компоненту.

JNDI ENC остается чрезвычайно важной частью соглашения между контейнером и компонентом в EJB 2.0. Хотя в главе 10 мы получали доступ к JDBC в постоянстве, реализуемом компонентом, посредством JNDI ENC, он не специфичен для объектных компонентов. JNDI ENC также используется сеансовыми компонентами, объектными компонентами и компонентами, управляемыми сообщениями. Чтобы избежать ненужного повторения, подробное обсуждение этого важного средства откладывается до главы 12. То, что вы узнаете из главы 12 об использовании JNDI ENC, одинаково применимо и к сеансовым компонентам, и к объектным компонентам, и к компонентам, управляемым сообщениями.

Жизненный цикл объектного компонента

Чтобы научиться лучше разрабатывать объектные компоненты, важно понять, как ими управляет контейнер. Спецификация EJB определяет почти все главные события в жизни объектного компонента, со времени его создания до времени его удаления «сборщиком мусора». Они называются *жизненным циклом* (*life cycle*), который дает разработчику компонента и производителям EJB всю необходимую им информацию по разработке компонентов и серверов EJB, придерживающихся последовательного протокола. Для того чтобы понять жизнен-

ный цикл, мы проследим путь экземпляра объекта через некоторые события жизненного цикла и опишем, как контейнер взаимодействует с объектным компонентом во время этих событий. На рис. 11.2 показан жизненный цикл экземпляра объекта.

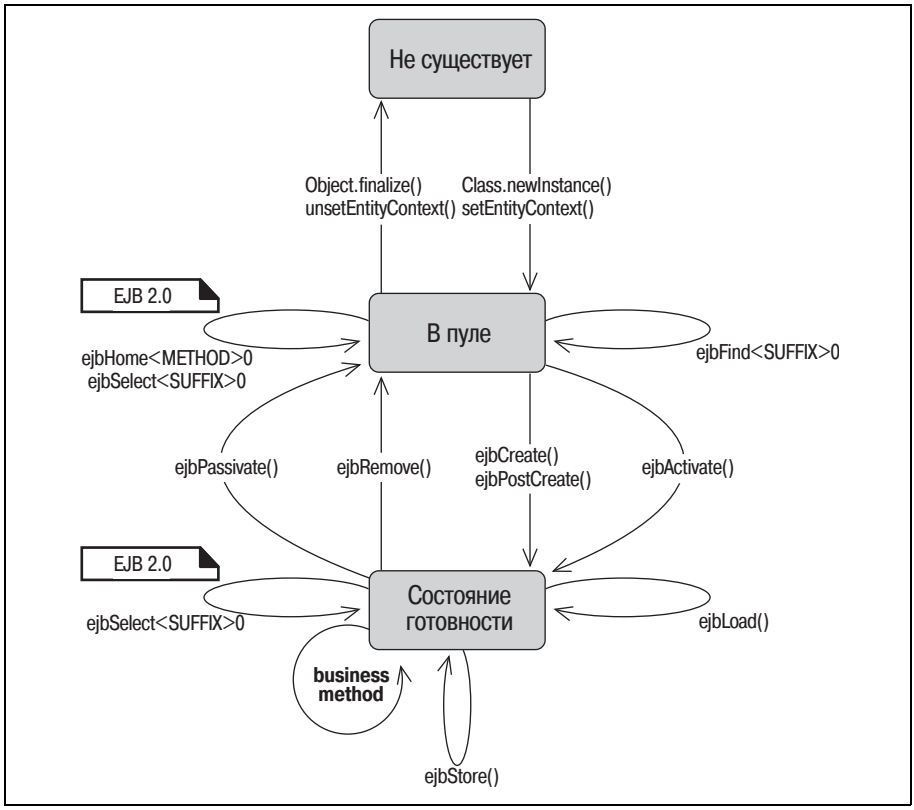

Рис. 11.2. Жизненный цикл объектного компонента

Этот раздел указывает пункты, в которых контейнер вызывает каждый из методов, описанных в интерфейсе `EntityBean`, а также поисковые методы и, для EJB 2.0, методы выборки и внутренние методы. Экземпляры компонента обязаны реализовывать интерфейс `EntityBean`, а это означает, что обращения методов обратного вызова являются обращениями к самому экземпляру компонента.

На каждом этапе жизненного цикла объектного компонента контейнер компонента предоставляет различные уровни доступа. Например, метод `EntityContext.getPrimary()` не будет работать, если его вызвать в методе `ejbCreate()`, но работает, когда вызывается из метода `ejbPostCreate()`. Другие методы `EJBContext` имеют похожие ограничения, как задано в JNDI ENC. В данном разделе говорится лишь о доступности этих методов, а в приложении B приведена таблица «Разрешенные

операции», содержащая подробную информацию о каждом методе класса компонента (`ejbCreate()`, `ejbActivate()`, `ejbLoad()` и т. д.).

Не существует

Объектный компонент начинает жизнь в виде набора файлов. В эту коллекцию входят дескриптор развертывания компонента, компонентные интерфейсы и все вспомогательные классы, сгенерированные во время развертывания. На этом этапе не существует ни одного экземпляра компонента.

В пуле

Во время запуска сервер EJB читает файлы компонентов и создает несколько экземпляров класса объектного компонента, который он помещает в пул. Экземпляры создаются вызовом метода `Class.newInstance()` класса компонента. Метод `newInstance()` создает экземпляр с помощью конструктора по умолчанию, не имеющего параметров.[1] Это означает, что поля постоянства экземпляров компонента устанавливаются в значения по умолчанию; сами экземпляры не представляют никаких данных в базе данных.

Сразу после создания экземпляра и непосредственно перед тем, как он будет помещен в пул, контейнер назначает экземпляру его `EntityContext`. `EntityContext` назначается с помощью вызова метода `setEntityContext()` интерфейса `EntityBean`, реализованного классом компонента. После того как экземпляру назначен его контекст, он помещается в пул экземпляров.

В пуле экземпляров экземпляр компонента доступен контейнеру в качестве возможного кандидата на обслуживание клиентских запросов. Однако пока он не будет запрошен, экземпляр компонента остается неактивным, если только он не используется для обслуживания методов запроса (т. е. поисковых или методов выборки) или запросов `ejbHome()`. Экземпляры компонента, находясь в пуле (pooled state), обычно используются для обслуживания запросов и `ejbHome()`, что имеет смысл, т. к. компоненты свободны, а данные методы не полагаются на состояние экземпляра компонента. Все экземпляры в пуле эквивалентны. Ни один из экземпляров не связан с компонентным объектом, и ни один из них не имеет значимого состояния.

Состояние готовности

Экземпляр компонента, находящийся в состоянии готовности (ready state), может принимать запросы клиента. Экземпляр компонента пере-

[1] Конструкторы не следует определять в классе компонента. Контейнеру должен быть доступен конструктор по умолчанию, не содержащий аргументов.

ходит в состояние готовности, когда контейнер связывает его с компонентным объектом. Это происходит в двух случаях: при создании нового объектного компонента и когда контейнер активирует объект.

Переход из пула в состояние готовности через создание

Когда клиентское приложение вызывает метод `create()` внутреннего объекта, прежде чем контейнер EJB сможет вернуть клиенту удаленную или локальную ссылку (компонентный объект), он должен выполнить некоторые действия. Для начала на сервере EJB должен быть создан компонентный объект.[1] После его создания экземпляр объектного компонента берется из пула экземпляров и связывается с этим компонентным объектом. Затем метод `create()`, вызванный клиентом, перенаправляется соответствующему ему методу `ejbCreate()` экземпляра компонента. После завершения метода `ejbCreate()` создается первичный ключ.

По окончании работы метода `ejbCreate()` вызывается метод `ejbPostCreate()` экземпляра объектного компонента. Наконец, после успешного завершения метода `ejbPostCreate()` внутренний компонент может вернуть клиенту удаленную или локальную ссылку – компонентный объект. Теперь экземпляр компонента и компонентный объект готовы к обслуживанию вызовов методов от клиента. Это – один способ, с помощью которого экземпляр компонента может быть перемещен из пула в состояние готовности.

Переход из пула в состояние готовности через метод запроса

Во время выполнения метода запроса каждый компонентный объект, найденный в результате запроса, будет создан с помощью перевода экземпляра из пула в состояние готовности. Когда объектный компонент найден, он связывается с компонентным объектом, а его ссылка на компонентный объект возвращается клиенту. Найденный компонент следует тому же протоколу, что и пассивируемый компонент; он активируется, когда клиент вызывает прикладной метод, и будет переведен в состояние готовности посредством процесса активации, как описано в следующем разделе.

Во многих случаях (в зависимости от производителя EJB) найденный объектный компонент в действительности не переводится в состояние готовности, пока он не запрашивается клиентом. Так, если поисковый метод возвращает коллекцию объектных компонентов, эти компоненты могут не активироваться, пока они не будут извлечены из кол-

[1] Это лишь концептуальная модель. В действительности контейнер EJB и компонентный объект могут являться одним и тем же объектом, или один компонентный объект может предоставлять несколько услуг для всех объектов этого типа. Эти детали реализации несущественны для понимания протокола жизненного цикла.

лекции или пока к ним не будет обращений со стороны клиента. Отложенная (при необходимости) активация объектных компонентов позволяет сэкономить системные ресурсы.

Переход из пула в состояние готовности через активацию

Процесс активации также может переместить экземпляр объектного компонента из пула в состояние готовности. Активация облегчает управление ресурсами, позволяя нескольким экземплярам компонента обслуживать многочисленные компонентные объекты. Активацию мы рассмотрели в главе 3, но здесь мы еще раз вспомним этот процесс, т. к. он касается жизненного цикла экземпляра объектного компонента. Активация предполагает, что объектный компонент перед этим был *пассивирован* (*passivated*). Об этом переходе состояния подробно рассказывается далее, а сейчас ограничимся высказыванием о том, что пассивируемый экземпляр компонента освобождает все ненужные ресурсы и оставляет компонентный объект, переходя в пул экземпляров. Когда экземпляр компонента возвращается в пул, компонентный объект остается без экземпляра, которому следует перенаправлять клиентские запросы. Компонентный объект поддерживает связь со своей заглушкой на стороне клиента – необходимо, чтобы объектный компонент не изменился, пока он нужен клиенту. Когда клиент вызывает прикладной метод компонентного объекта, компонентный объект должен получить новый экземпляр компонента. Это выполняется с помощью активации экземпляра компонента.

Активированный экземпляр компонента покидает пул экземпляров и назначается компонентному объекту. После того как он назначен подходящему компонентному объекту, вызывается его метод `ejbActivate()`; теперь `EntityContext` экземпляра может предоставлять информацию, специфичную для компонентного объекта, но не может предоставлять информацию о безопасности и транзакциях. Метод обратного вызова `ejbActivate()` экземпляра компонента может применяться для повторного получения ресурсов или выполнения любой другой необходимой работы перед обслуживанием клиентов.

При активировании экземпляра объектного компонента непостоянные поля экземпляра компонента могут содержать произвольные («грязные») значения и поэтому должны быть повторно проинициализированы в методе `ejbActivate()`.

В постоянстве, управляемом контейнером, управляемые контейнером поля автоматически синхронизируются с базой данных после вызова `ejbActivate()`, но до обслуживания экземпляром компонента прикладных методов. Порядок, в котором эти события происходят в CMP-объектных компонентах:

1. Вызывается метод `ejbActivate()` экземпляра компонента.
2. Автоматически синхронизируются поля постоянства.

3. Метод `ejbLoad()` уведомляет компонент о том, что его поля постоянства были синхронизированы.
4. Вызываются необходимые прикладные методы.

В постоянстве, управляемом компонентом, поля постоянства синхронизируются методом `ejbLoad()` после вызова `ejbActivate()`, но перед тем как смогут вызываться прикладные методы. Порядок действий в постоянстве, управляемом компонентом, следующий:

1. Вызывается `ejbActivate()` экземпляра компонента.
2. Вызывается `ejbLoad()`, для того чтобы дать компоненту возможность синхронизировать его поля постоянства.
3. Вызываются необходимые прикладные методы.

Переход из состояния готовности в пул через пассивацию

Компонент может перейти из состояния готовности в пул посредством пассивации, представляющей собой процесс отсоединения экземпляра компонента от компонентного объекта, когда он свободен. После того как экземпляр компонента будет связан с компонентным объектом, контейнер EJB может в любое время пассивировать данный экземпляр при условии, что он в настоящее время не обрабатывает вызов метода. В ходе процесса пассивации для экземпляра компонента вызывается метод `ejbPassivate()`. Этот метод обратного вызова может использоваться экземпляром для освобождения ресурсов или выполнения другой обработки перед отсоединением от компонентного объекта. По завершении `ejbPassivate()` экземпляр компонента отсоединяется от серверного компонентного объекта и возвращается в пул экземпляров. Теперь экземпляр компонента возвращен в состояние нахождения в пуле.

Управляемый компонентом экземпляр объекта не должен пытаться в методе `ejbPassivate()` сохранить свое состояние в базе данных. Это действие должно выполняться в методе `ejbStore()`. Перед пассивацией компонента контейнер вызывает `ejbStore()` для синхронизации состояния экземпляра компонента с базой данных.

Основное, о чем следует помнить – для объектных компонентов пассивация является просто уведомлением о том, что экземпляр скоро будет отсоединен от компонентного объекта. В отличие от сеансовых компонентов с состоянием, поля экземпляра объектного компонента – не сериализуемые и при пассивации компонента сохраняются внутри компонентного объекта. Какие бы значения ни хранились в непостоянных полях экземпляра, при связывании объектного компонента с компонентным объектом эти значения будут перенесены в компонентном объекте до следующего связывания.

Переход из состояния готовности в пул через удаление

Экземпляр компонента перемещается из состояния готовности в пул также и при его удалении. Это происходит, когда клиентское приложение вызывает один из удаляющих методов компонентного или внутреннего объекта. В случае объектных компонентов вызов удаляющего метода стирает данные объекта из базы данных. Как только данные объекта будут удалены из базы данных, он перестанет быть действительным.

После завершения метода `ejbRemove()` экземпляр компонента перемещается обратно в пул экземпляра и выходит из состояния готовности. Важно, что метод `ejbRemove()` освобождает любые ресурсы, которые обычно освобождаются методом `ejbPassivate()`, не вызываемым при удалении компонента. При необходимости это может быть сделано с помощью вызова метода `ejbPassivate()` из тела метода `ejbRemove()`.

В постоянстве, управляемом компонентом, метод `ejbRemove()` реализуется разработчиком объектного компонента и содержит код для удаления данных компонента из базы данных. Контейнер EJB вызывает метод `ejbRemove()` в ответ на вызов клиентом метода `remove()` одного из компонентных интерфейсов.

В постоянстве, управляемом контейнером, метод `ejbRemove()` уведомляет экземпляр объектного компонента, что он собирается удалить данные компонента из базы данных. Сразу же после вызова `ejbRemove()` контейнер удаляет данные объектного компонента.

В EJB 2.0 CMP-контейнер также разрывает отношения компонента с другими объектными компонентами в базе данных. Если для отношения определено каскадное удаление, контейнер удаляет все относящиеся к нему объектные компоненты. Это включает активацию каждого компонента и вызов его метода `ejbActivate()`, загрузку состояния для каждого компонента с помощью вызова метода `ejbLoad()`, вызов `ejbRemove()` для всех компонентов в отношении и последующее удаление их данных. Цепное выполнение этого процесса может продолжаться, пока не будут завершены все операции каскадного удаления для всех отношений.

Жизнь в состоянии готовности

Компонент находится в состоянии готовности, пока он связан с компонентным объектом и готов обслуживать запросы от клиента. Когда клиент вызывает прикладной метод, такой как `Ship.getName()` удаленной или локальной ссылки компонента (компонентного объекта), вызов метода принимается сервером EJB и перенаправляется экземпляру компонента. Экземпляр выполняет метод и возвращает результат. Пока экземпляр компонента находится в состоянии готовности, он может обслуживать все вызываемые клиентом прикладные методы.

Прикладные методы могут вызываться несколько раз в любом порядке, а могут вообще не вызываться.

Кроме обслуживания прикладных методов, находясь в состоянии готовности, объектный компонент может выполнять методы выборки, вызываемые экземпляром компонента, обслуживающим прикладной метод.

Методы `ejbLoad()` и `ejbStore()`, синхронизирующие состояние экземпляра компонента с базой данных, могут вызываться только тогда, когда компонент находится в состоянии готовности. Эти методы могут вызываться в любом порядке, в зависимости от реализации производителя. Некоторые производители вызывают `ejbLoad()` перед каждым обращением метода и `ejbStore()` после каждого обращения метода, в зависимости от контекста транзакции. У других производителей эти методы вызываются не так часто.

В постоянстве, управляемом компонентом, метод `ejbLoad()` для получения данных из базы данных всегда должен использовать метод `EntityContext.getPrimaryKey()` и не должен полагаться ни на первичный ключ, ни на другие данные, которые компонент оставил в его полях. (Именно так мы реализовали это в управляемой компонентом версии компонента Ship в главе 10.) Однако следует полагать, что состояние компонента при вызове метода `ejbStore()` является действительным.

В постоянстве, управляемом контейнером, метод `ejbLoad()` всегда вызывается сразу после синхронизации управляемых контейнером полей компонента с базой данных – другими словами, сразу после того, как контейнер обновит состояние экземпляра компонента в базе данных. Это дает возможность выполнить любые вычисления или преобразования данных перед тем, как экземпляр начнет обслуживать вызовы прикладных методов от клиентов. Метод `ejbStore()` вызывается непосредственно перед тем, как база данных будет синхронизироваться с состоянием экземпляра компонента – непосредственно перед тем, как контейнер запишет управляемые им поля в базу данных. Это дает CMP-экземпляру компонента возможность изменить данные в управляемых контейнером полях перед их сохранением в базе данных.

В постоянстве, управляемом компонентом, методы `ejbLoad()` и `ejbStore()` вызываются тогда, когда контейнер считает, что настало время для синхронизации состояния компонента с базой данных. Они являются единственными методами обратного вызова, которые должны применяться для синхронизации состояния компонента с базой данных. Для обращения к базе данных с целью синхронизации не пользуйтесь методами `ejbActivate()`, `ejbPassivate()`, `setEntityContext()` и `unsetEntityContext()`. Следует предпочесть методы `ejbCreate()` и `ejbRemove()` для вставки и удаления (соответственно) данных объекта из базы данных.

Окончание жизненного цикла

Жизненный цикл экземпляра компонента заканчивается, когда контейнер решает удалить его из пула и разрешить сборщику мусора «собрать его». Это случается при различных обстоятельствах. Если контейнер решает уменьшать число экземпляров в пуле – обычно для экономии ресурсов, – он освобождает один или несколько экземпляров компонентов и позволяет им быть удаленными сборщиком мусора. Способность изменять размер пула экземпляров позволяет серверу EJB управлять своими ресурсами (количеством потоков, доступной памятью и т. д.) так, чтобы можно было достичь максимально высокой эффективности.

Когда сервер EJB заканчивает работу, большинство контейнеров освобождают все экземпляры компонента так, чтобы они могли быть безопасно собраны сборщиком мусора. Некоторые контейнеры также могут решить освободить любое количество экземпляров, которые ведут себя неправильно или пострадали от какой-то невосстановимой ошибки, сделавшей их работу нестабильной. Например, в любое время, когда экземпляр объектного компонента в любом своем методе генерирует тип исключения `RuntimeException`, контейнер EJB удаляет этот экземпляр из памяти и заменяет его устойчивым экземпляром из пула экземпляров.

Когда экземпляр объектного компонента покидает пул экземпляров для того, чтобы быть собранным сборщиком мусора, контейнер вызывает метод `unsetEntityContext()` для уведомления экземпляра компонента о том, что он скоро будет разрушен. Данный метод обратного вызова позволяет экземпляру компонента освободить любые ресурсы, которые он удерживал, перед тем как быть собранным сборщиком мусора. После завершения метода `unsetEntityContext()` компонент удаляется из памяти.

После метода `unsetEntityContext()` может быть вызван, а может и не вызываться, метод `finalize()` экземпляра компонента. Компонент не должен рассчитывать на вызов его метода `finalize()`, т. к. у каждого производителя удаление экземпляров обрабатывается по-разному.

12

Сеансовые компоненты

Как было показано в главах 6–11, объектные компоненты предоставляют объектно-ориентированный интерфейс, упрощающий разработчикам создание, изменение и удаление данных в базе данных. Объектные компоненты позволяют разработчикам повысить производительность за счет поддержки многократного использования кода и сокращают стоимость разработки. Можно многократно обращаться к понятию Ship во всей прикладной системе без необходимости переопределять, переписывать и повторно тестировать прикладную логику и доступ к данным.

Однако объектные компоненты – это еще не все. Кроме них мы видели еще один тип компонентов – сеансовые. Сеансовые компоненты заполняют пробелы, оставленные объектными компонентами. Они полезны для описания взаимодействия между другими компонентами (рабочий поток) и для реализации определенных задач. В отличие от объектных компонентов, сеансовые не представляют общедоступные данные в базе данных, но они могут к ним обращаться. Это означает, что с помощью сеансовых компонентов мы можем организовать чтение, модификацию и добавление данных. Например, мы могли бы использовать сеансовый компонент для предоставления информационных списков, например списка всех доступных кают. Иногда мы можем генерировать список, взаимодействуя с объектными компонентами, как, например, список кают, который мы разработали для компонента TravelAgent в главе 4. Но чаще сеансовые компоненты генерируют списки, обращаясь непосредственно к базе данных.

Итак, когда для непосредственного доступа к данным следует применять объектный компонент, а когда – сеансовый? Хороший вопрос! Вот практическое правило – объектный компонент разрабатывается для предоставления безопасного и логичного интерфейса к набору общедоступных данных, описывающему некоторое понятие. Эти данные могут часто изменяться. Сеансовые компоненты обращаются к данным, которые охватывают понятие, они не используются совместно и обычно предназначены только для чтения.

Кроме непосредственного доступа к данным сеансовые компоненты могут представлять *рабочий поток* (*workflow*). Рабочий поток описывает все шаги, необходимые для выполнения определенной задачи, такой как заказ билетов на корабль или прокат видео. Сеансовые компоненты являются частью того же прикладного API, что и объектные компоненты, но как компоненты рабочего потока они служат другой цели. Сеансовые компоненты могут управлять взаимодействием между объектными компонентами, описывая то, как они должны совместно работать, чтобы выполнить определенную задачу. Отношения между сеансовыми компонентами и объектными походит на отношения между сценарием пьесы и актерами, которые ее исполняют. Здесь объектные компоненты являются актерами, а сеансовый компонент – сценарием. Актеры без сценария могут самостоятельно выполнять каждый свою функцию, но рассказать историю они могут только в контексте сценария. В терминах нашего примера не имеет смысла заполнять базу данных информацией о каютах, судах, клиентах и других объектах, если нельзя создать взаимоотношения между ними, например зарегистрировать клиента на определенный круиз.

Сеансовые компоненты делятся на два основных типа: без состояния и с состоянием. Сеансовый компонент *без состояния* (*stateless*) – это коллекция связанных услуг, каждая из которых представлена методом. Компонент не поддерживает никакого состояния в промежутках между обращениями к методу. Когда вызывается метод сеансового компонента без состояния, компонент выполняет метод и возвращает результат, не зная и не заботясь о запросах, приходивших до него и, возможно, следующих за ним. Сеансовый компонент без состояния можно себе представить как набор процедур или пакетных программ, которые выполняют запрос, основанный на некоторых параметрах, и возвращают результат. Сеансовые компоненты без состояния, как правило, выступают в качестве компонентов общего назначения или многократного использования, таких как программные службы.

Сеансовый компонент *с состоянием* (*stateful*) – это расширение клиентского приложения. Он выполняет задачи от имени клиента и поддерживает состояние, связанное с этим клиентом. Это состояние называется *состоянием диалога* (*conversational state*) из-за того, что оно представляет собою непрерывный диалог между сеансовым компонентом с состоянием и клиентом. Методы, вызываемые для сеансового

компонента с состоянием, могут писать и читать данные из этого состояния диалога, которое одновременно доступно для всех методов компонента. Сеансовые компоненты с состоянием, как правило, специализируются под конкретный сценарий. Они предоставляют логику, которая может быть размещена в клиентском приложении двухуровневой системы. Сеансовые компоненты (с состоянием или без) не являются постоянными, как объектные компоненты. Другими словами, сеансовые компоненты не хранятся в базе данных.

В зависимости от производителя, сеансовые компоненты с состоянием могут иметь определенное время действия (тайм-аут). Если клиент откажется от применения компонента с состоянием до того, как это время истечет, экземпляр компонента разрушается, а ссылка на компонентный объект становится недействительной. Это не дает сеансовому компоненту с состоянием «терять время даром» в случае, если клиент отключился или каким-либо другим образом закончил с ним работать. Клиент может также явно удалить сеансовый компонент с состоянием, вызвав один из его удаляющих методов.

Сеансовые компоненты без состояния живут дольше, т. к. не хранят состояние диалога и не выделяются конкретному клиенту, но они и не сохраняются в базе данных, потому что не представляют никакие данные. Как только сеансовый компонент без состояния закончил обработку вызова метода для клиента, он может быть переназначен любому другому компонентному объекту для обслуживания нового клиента. Клиент может поддерживать связь с компонентным объектом сеансового компонента без состояния, но сам экземпляр компонента может обслуживать запросы любого клиента. Из-за того что они не содержат никакой информации о состоянии, сеансовые компоненты без состояния неразличимы для клиентов. Сеансовые компоненты без состояния также могут иметь определенное время действия и могут быть удалены клиентом, но результат этих событий отличается от того, который был у сеансовых компонентов с состоянием. В случае с сеансовым компонентом без состояния тайм-аут (или операция удаления) просто делает ссылку на компонентный объект для этого клиента недействительной. Экземпляр компонента не разрушается и может обслуживать другие клиентские запросы.

Сеансовый компонент без состояния

Сеансовый компонент без состояния весьма эффективен и относительно легок в реализации. Сеансовые компоненты без состояния требуют мало серверных ресурсов, потому что они не являются ни постоянными, ни выделенными конкретному клиенту. Поскольку они не выделяются клиенту, относительно небольшое число экземпляров компонентов без состояния могут одновременно использоваться многими компонентными объектами. Сеансовый компонент без состояния не

поддерживает состояние диалога, относящегося к клиенту, которого он обслуживает, поэтому он может свободно передаваться между компонентными объектами. Как только экземпляр без состояния заканчивает обслуживание вызова метода, он может перейти к другому компонентному объекту. Из-за того что он не поддерживает состояние диалога, сеансовый компонент без состояния не требует ни пассивации, ни активации, еще больше уменьшая затраты на переход. Короче говоря, сеансовые компоненты без состояния – легкие и быстрые.

Сеансовые компоненты без состояния часто предоставляют услуги достаточно универсальные и пригодные для многократного использования. Услуги могут быть связаны, но они не являются взаимозависимыми. Это означает, что все, что необходимо знать методу, должно быть передано через его параметры. Что приводит к любопытному ограничению. Сеансовые компоненты без состояния не могут ничего помнить в промежутках между вызовами методов. Следовательно, они вынуждены выполнять всю задачу за один вызов метода. Единственное исключение из этого правила представляет информация, доступная из `SessionContext` и JNDI ENC.

Сеансовые компоненты без состояния представляют собой EJB-версии традиционных прикладных программ обработки транзакций, выполняемых с помощью вызова процедур. Процедура выполняется с начала до конца, а затем возвращает результат. Как только процедура завершена, от обрабатываемых ею данных и деталей запроса ничего не остается. Отсутствует всякое состояние.

Эти ограничения не означают, что у сеансового компонента без состояния не может быть своих переменных экземпляра, и он не может поддерживать никакого внутреннего состояния. Никто не запрещает хранить переменную, которая отслеживает количество вызовов компонента или сохраняет данные для отладочных целей. Переменная экземпляра может даже поддерживать ссылку на действующий ресурс, такой как подключения URL для регистрации, система проверки кредитных карточек или какой-либо другой, который может быть полезен. Ресурс должен быть получен через JNDI ENC. Однако важно помнить, что клиент никогда не видит это состояние. Клиент не должен исходить из предположения, что каждый раз его обслуживает один и тот же экземпляр компонента. Если эти переменные в разных экземплярах компонента имеют разные значения, то по мере переназначения их от одного клиента другому будет казаться, что их значения меняются произвольно. Поэтому любые ресурсы, на которые мы ссылаемся в переменных экземпляра, должны быть универсальными. Например, каждый экземпляр компонента может при необходимости делать записи отладочных сообщений, что может оказаться единственным способом выяснить, что происходит на достаточно большом сервере, содержащем множество экземпляров компонентов. Клиент не знает и не хочет знать о том, как происходит запись отладочных сооб-

щений. Однако совершенно неприемлемо для компонента без состояния было бы помнить, что он участвовал в процессе заказа билетов для Мадам X – в следующий раз, когда он будет вызван, он может обслуживать совершенно другого клиента.

Сеансовые компоненты без состояния могут применяться для создания отчетов, пакетной обработки или предоставления услуг, не требующих состояния, таких как проверка достоверности кредитных карточек. Еще одним удачным примером может служить компонент StockQuote, возвращающий текущую цену акций. Любое действие, которое может быть выполнено в одном запросе метода, является хорошим кандидатом на реализацию в виде высокоэффективного сеансового компонента без состояния.

Компонент ProcessPayment

В главах 2 и 3 рассматривался компонент TravelAgent, который содержит прикладной метод с именем `bookPassage()`, использующий компонент ProcessPayment. В следующем разделе разрабатывается законченное определение компонента TravelAgent, включая логику метода `bookPassage()`. Однако в настоящий момент нас больше всего интересует компонент ProcessPayment, представляющий собою компонент без состояния, который используется компонентом TravelAgent для взимания с клиентов платы за круиз.

Взимание платы с клиентов – обычное действие в прикладных системах «Титан». Взимать плату с клиентов требуется не только в системе резервирования, но и в магазинах подарков «Титана», бутиках и других сопутствующих предприятиях. Процесс взимания платы с клиента за услуги является обычным во многих системах, поэтому он должен быть инкапсулирован в свой собственный компонент.

Платежи регистрируются в специальной таблице базы данных с именем `PAYMENT`. Данные из `PAYMENT` обрабатываются в пакетном режиме с целью учета и обычно не используются вне его. Другими словами, эти данные в системе «Титан» лишь добавляются; они не читаются, не модифицируются и не удаляются. Поскольку процесс взимания платы может быть выполнен в одном методе, и из-за того что данные не модифицируются часто и не используются совместно, для обработки платежей мы выберем сеансовый компонент без состояния. Возможны несколько форм оплаты: кредитными карточками, чеками или наличными. В нашем компоненте без состояния ProcessPayment мы будем моделировать все эти формы оплаты.

PAYMENT: Таблица базы данных

Компонент ProcessPayment обращается к существующей в системе «Титан» таблице с именем `PAYMENT`. Создайте в базе данных таблицу с именем `PAYMENT` со следующим определением:

```
CREATE TABLE PAYMENT
(
    customer_id     INTEGER,
    amount          DECIMAL(8,2),
    type            CHAR(10),
    check_bar_code  CHAR(50),
    check_number    INTEGER,
    credit_number   CHAR(20),
    credit_exp_date DATE
)
```

ProcessPaymentRemote: Удаленный интерфейс

Как и любому другому компоненту, сеансовому компоненту без состояния необходим компонентный интерфейс. Хотя в EJB 1.1 используются только удаленные интерфейсы, в EJB 2.0 у сеансовых компонентов могут быть и локальный, и удаленный интерфейсы.

Очевидно, что в удаленном интерфейсе нам понадобится метод `byCredit()`, из-за того что его использует компонент TravelAgent. Также мы можем выделить два других метода, которые будут нам необходимы: `byCash()` – для клиентов, расплачивающихся наличными, и `byCheck()` – для клиентов, оплачивающих услуги персональным чеком.

Приведем законченное определение удаленного интерфейса для компонента ProcessPayment:

```
package com.titan.processpayment;

import java.rmi.RemoteException;
import java.util.Date;
import com.titan.customer.CustomerRemote;

public interface ProcessPaymentRemote extends javax.ejb.EJBObject {

public boolean byCheck(CustomerRemote customer, CheckDO check, double amount)
        throws RemoteException,PaymentException;

    public boolean byCash(CustomerRemote customer, double amount)
        throws RemoteException,PaymentException;

    public boolean byCredit(CustomerRemote customer, CreditCardDO card,
        double amount) throws RemoteException,PaymentException;
}
```

Удаленные интерфейсы в сеансовых компонентах подчиняются тем же самым правилами, что и в объектных компонентах. Здесь мы определили три прикладных метода: `byCheck()`, `byCash()` и `byCredit()`, принимающих информацию, необходимую для выбранной формы оплаты, и возвращающих значение типа `boolean`, указывающее на успешность выполнения платежа. В дополнение к обязательному `RemoteException` эти методы могут генерировать специфическое для данного приложения исключение – `PaymentException`. Последнее генерируется,

если при обработке оплаты имеют место какие-то проблемы, такие как неверный номер чека или истекший срок годности кредитной карточки. Однако заметьте, что в интерфейсе ProcessPaymentRemote нет ничего специфичного для системы заказов. В системе «Титан» он мог бы применяться везде. Кроме того, каждый метод, определенный в удаленном интерфейсе, полностью независим от остальных. Все данные, необходимые для обработки платежа, поступают через параметры методов.

Как расширение интерфейса javax.ejb.EJBObject удаленный интерфейс сеансового компонента наследует те же самые функциональные возможности, что и удаленный интерфейс объектного компонента. Однако метод getPrimaryKey() генерирует RemoteException, т. к. у сеансовых компонентов нет первичного ключа, который можно было бы вернуть:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome() throws RemoteException;
    public abstract Handle getHandle() throws RemoteException;
    public abstract Object getPrimaryKey() throws RemoteException;
    public abstract boolean isIdentical(EJBObject obj) throws
RemoteException;
    public abstract void remove() throws RemoteException, RemoveException;
}
```

Метод getHandle() возвращает сериализуемый объект Handle (дескриптор), точно так же, как и метод getHandle() объектного компонента. Для сеансовых компонентов без состояния этот дескриптор может быть сериализован и повторно использован в любое время, пока тип данного компонента все еще доступен в контейнере, создавшем дескриптор.

В отличие от сеансовых компонентов без состояния, компоненты с состоянием доступны через Handle, только пока этот отдельный экземпляр компонента остается действующим на сервере EJB. Если клиент явно уничтожает сеансовый компонент с состоянием с помощью одного из методов remove() либо истекает срок действия компонента, то экземпляр разрушается, а Handle становится недействительным. Как только сервер удаляет сеансовый компонент с состоянием, его Handle больше не является действующим и при вызове его метода getEJBObject() будет генерироваться исключение RemoteException.

Можно получить удаленную ссылку на компонент через дескриптор Handle, вызвав его метод getEJBObject():

```
public interface javax.ejb.Handle {
    public abstract EJBObject getEJBObject() throws RemoteException;
}
```

Компонент ProcessPayment хранится в своем собственном пакете, и это означает, что у него есть свой собственный каталог *dev/com/titan/processpayment* в нашем рабочем дереве. В этом каталоге мы храним весь код для данного компонента и компилируем его файлы классов.

Зависимые объекты: Классы CreditCardDO и CheckDO

Удаленный интерфейс компонента ProcessPayment использует в своем определении два класса, представляющие особенный интерес: `CreditCardDO` и `CheckDO`. Определения этих классов следующие:

```
/* CreditCardDO.java */
package com.titan.processpayment;

import java.util.Date;

public class CreditCardDO implements java.io.Serializable {
    final static public String MASTER_CARD = "MASTER_CARD";
    final static public String VISA = "VISA";
    final static public String AMERICAN_EXPRESS = "AMERICAN_EXPRESS";
    final static public String DISCOVER = "DISCOVER";
    final static public String DINERS_CARD = "DINERS_CARD";

    public String number;
    public Date expiration;
    public String type;

    public CreditCardDO(String nmbr, Date exp, String typ) {
        number = nmbr;
        expiration = exp;
        type = typ;
    }
}

/* CheckDO.java */
package com.titan.processpayment;

public class CheckDO implements java.io.Serializable {
    public String checkBarCode;
    public int checkNumber;

    public CheckDO(String barCode, int number) {
        checkBarCode = barCode;
        checkNumber = number;
    }
}
```

`CreditCardDO` и `CheckDO` представляют собой зависимые объекты. Это понятие мы рассматривали в главе 6 на примере компонента Address. Посмотрев на определения классов `CreditCardDO` и `CheckDO`, вы увидите, что это не компоненты, а просто сериализуемые классы Java. Эти классы предоставляют удобный механизм для транспортировки и соединения связанных данных. `CreditCardDO`, например, группирует в од-

ном классе все данные, имеющие отношение к кредитной карточке, облегчая передачу этой информации через сеть, а также делая наши интерфейсы чуть понятнее.

PaymentException: Прикладное исключение

Любой удаленный или локальный интерфейс, принадлежит ли он объектному компоненту или сеансовому компоненту, может генерировать прикладные исключения. Прикладные исключения создаются разработчиком компонента и предназначены для описания проблемы в прикладной логике – в нашем случае ошибку при проведении платежа. Прикладные исключения должны иметь значение для клиента, давая краткое и понятное описание ошибки.

Важно понимать, какие исключения и когда следует использовать. `RemoteException` указывает на проблемы подсистемного уровня и используется средствами RMI. Точно так же исключения типа `javax.naming.NamingException` и `java.sql.SQLException` возбуждаются другими подсистемами Java. Как правило, они не должны явно генерироваться вашими компонентами. Транслятор Java требует, чтобы для перехвата подобных проверяемых им исключений применялись блоки `try/catch`.

В EJB 2.0 исключение `EJBException` может означать проблемы, с которыми встречается контейнер при обработке вызовов локальных интерфейсов. `EJBException` – это непроверяемое исключение, поэтому мы не получим ошибку компиляции, если не напишем код для ее обработки. Однако иногда `EJBException` лучше перехватывать, а иногда его следует передать дальше.

Когда метод компонента перехватывает исключение, пришедшее из подсистем (JDBC, JNDI, JMS и т. д.), оно должно быть повторно возбуждено в виде `EJBException` или прикладного исключения. Необходимо повторно сгенерировать перехватываемое исключение в виде `EJBException`, если оно означает ошибку на системном уровне. Перехватываемые исключения повторно возбуждаются в виде прикладных исключений, если они вызваны логическими ошибками приложения. Компоненты содержат прикладную логику, и если затруднение имеет место в прикладной логике, то его необходимо представить прикладным исключением. Когда компонент генерирует `EJBException` или какой-то другой тип `RuntimeException`, исключение сначала обрабатывается контейнером, который выбрасывает данный экземпляр компонента и заменяет его другим. Контейнер, обработав исключение, передает его клиенту. Для удаленных клиентов контейнер генерирует `RemoteException`, а для локальных (совмещенных) компонентов контейнер повторно генерирует первоначальное исключение `EJBException` или `RuntimeException`, возбужденное экземпляром компонента.

Исключение `PaymentException` описывает специфичную прикладную проблему, поэтому оно является прикладным. Прикладные исключе-

ния расширяют интерфейс java.lang.Exception. Любые переменные экземпляра, которые мы вводим в эти исключения, должны быть сериализуемыми.

Здесь приведено определение прикладного исключения PaymentException:

```
package com.titan.processpayment;

public class PaymentException extends java.lang.Exception {
    public PaymentException() {
        super();
    }
    public PaymentException(String msg) {
        super(msg);
    }
}
```

ProcessPaymentHomeRemote: Внутренний интерфейс

Внутренний интерфейс сеансового компонента без состояния должен объявить единственный метод create() без параметров. Этого требует спецификация EJB. Нельзя определять методы create(), содержащие параметры, из-за того что сеансовые компоненты без состояния не поддерживают состояние диалога, которое следует инициализировать. У сеансовых компонентов нет ни одного поискового метода, т. к. сеансовые компоненты не содержат первичных ключей и не представляют никаких данных в базе данных.

Хотя EJB 2.0 и определяет методы create<*SUFFIX*>() для сеансовых компонентов с состоянием и объектных компонентов, в сеансовых компонентах без состояния должен быть определен только один метод create() – без суффикса и без параметров. Это также справедливо и для EJB 1.1. Причина данного ограничения связана с жизненным циклом сеансовых компонентов без состояния, который рассматривается далее в этой главе.

Далее приведено определение удаленного внутреннего интерфейса для компонента ProcessPayment:

```
package com.titan.processpayment;

import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface ProcessPaymentHomeRemote extends javax.ejb.EJBHome {
    public ProcessPaymentRemote create() throws RemoteException,
CreateException;
}
```

Исключение CreateException является обязательным, как и RemoteException, и может быть возбуждено непосредственно компонентом для указания на прикладную ошибку при создании компонента. RemoteEx-

ception возбуждается при возникновении других системных ошибок, например, когда возникают проблемы с сетевой связью или когда классом компонента возбуждается неперехватываемое исключение.

Интерфейс ProcessPaymentHomeRemote как расширение javax.ejb.EJBHome содержит те же самые методы EJBHome, что и объектные компоненты. Единственное различие состоит в том, что не работает метод remove(Object primaryKey), поскольку сеансовые компоненты не имеют первичных ключей. Если для сеансового компонента (без состояния или с состоянием) вызывается метод EJBHome.remove(Object primaryKey), то возбуждается исключение RemoteException. По логике, этот метод никогда не должен вызываться для удаленного внутреннего интерфейса сеансового компонента.

Ниже приведены определения интерфейсов javax.ejb.EJBHome для EJB 1.1 и 2.0:

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {
    public abstract HomeHandle getHomeHandle() throws RemoteException;
    public abstract EJBMetaData getEJBMetaData() throws RemoteException;
    public abstract void remove(Handle handle) throws RemoteException,
        RemoveException;
    public abstract void remove(Object primaryKey) throws RemoteException,
        RemoveException;
}
```

Точно так же, как и для объектного компонента, внутренний интерфейс сеансового компонента может возвращать относящийся к компоненту объект EJBMetaData – сериализуемый объект, который предоставляет информацию об интерфейсах компонента. Единственное различие между EJBMetaData для сеансового компонента и объектного компонента состоит в том, что при вызове getPrimaryKeyClass() в EJBMetaData сеансового компонента генерируется исключение java.lang.RuntimeException:

```
public interface javax.ejb.EJBMetaData {
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
    public abstract Class getPrimaryKeyClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
    public abstract boolean isStateless();  // только для EJB 1.0
}
```

ProcessPaymentBean: Класс компонента

Как было сказано раньше, компонент ProcessPayment обращается к данным, которые обычно не используются совместно, поэтому он является превосходным кандидатом на реализацию в виде сеансового компонента без состояния. Этот компонент в действительности представляет набор независимых операций – еще один признак того, что он хороший кандидат на статус сеансового компонента без состояния.

Здесь приведено определение класса `ProcessPaymentBean`**, поддерживающего функциональные возможности удаленного интерфейса:**

```
package com.titan.processpayment;
import com.titan.customer.*;

import java.sql.*;
import java.rmi.RemoteException;
import javax.ejb.SessionContext;

import javax.naming.InitialContext;
import javax.sql.DataSource;
import javax.ejb.EJBException;
import javax.naming.NamingException;

public class ProcessPaymentBean implements javax.ejb.SessionBean {

    final public static String CASH = "CASH";
    final public static String CREDIT = "CREDIT";
    final public static String CHECK = "CHECK";

    public SessionContext context;

    public void ejbCreate() {
    }

    public boolean byCash(CustomerRemote customer, double amount)
        throws PaymentException{
       return process(getCustomerID(customer), amount, CASH, null, -1, null,
           null);
    }

    public boolean byCheck(CustomerRemote customer, CheckDO check, double
                           amount)
        throws PaymentException{
        int minCheckNumber = getMinCheckNumber();
        if (check.checkNumber > minCheckNumber) {
            return process(getCustomerID(customer), amount, CHECK,
                check.checkBarCode, check.checkNumber, null, null);
        }
        else {
            throw new PaymentException("Check number is too low.
                Must be at least "+minCheckNumber);
        }
    }
    public boolean byCredit(CustomerRemote customer, CreditCardDO card,
        double amount) throws PaymentException {
        if (card.expiration.before(new java.util.Date())) {
            throw new PaymentException("Expiration date has"+ " passed");
        }
        else {
            return process(getCustomerID(customer), amount, CREDIT,
```

```
            null, -1, card.number,
            new java.sql.Date(card.expiration.getTime()));
        }
    }
    private boolean process(Integer customerID, double amount, String type,
        String checkBarCode, int checkNumber, String creditNumber,
        java.sql.Date creditExpDate) throws PaymentException {

        Connection con = null;

        PreparedStatement ps = null;

        try {
            con = getConnection();
            ps = con.prepareStatement
                ("INSERT INTO payment (customer_id, amount, type,"+
                "check_bar_code,check_number,credit_number,"+
                "credit_exp_date) VALUES (?,?,?,?,?,?,?)");
            ps.setInt(1,customerID.intValue());
            ps.setDouble(2,amount);
            ps.setString(3,type);
            ps.setString(4,checkBarCode);
            ps.setInt(5,checkNumber);
            ps.setString(6,creditNumber);
            ps.setDate(7,creditExpDate);
            int retVal = ps.executeUpdate();
            if (retVal!=1) {
                throw new EJBException("Payment insert failed");
            }
            return true;
        } catch(SQLException sql) {
            throw new EJBException(sql);
        } finally {
            try {
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se) {
                se.printStackTrace();
            }
        }
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        context = ctx;
    }
    private Integer getCustomerID(CustomerRemote customer) {
        try {
            return (Integer)customer.getPrimaryKey();
```

```
            } catch(RemoteException re) {
                throw new EJBException(re);
            }
        }
        private Connection getConnection() throws SQLException {
            // реализация показана ниже
        }
        private int getMinCheckNumber() {
            // реализация показана ниже
        }
    }
```

Все три способа оплаты используют закрытый вспомогательный метод `process()`, выполняющий работу по добавлению платежа в базу данных. Эта стратегия уменьшает возможность ошибки программиста и упрощает сопровождение компонента. Метод `process()` просто вставляет информацию о платеже в таблицу `PAYMENT`. Применение JDBC в данном методе должно быть вам знакомо по работе над компонентом Ship в главе 10. Соединение с JDBC получено посредством метода `getConnection()`, как показано в следующем коде:

```
    private Connection getConnection() throws SQLException {
        try {
            InitialContext jndiCntx = new InitialContext();
            DataSource ds = (DataSource)
                jndiCntx.lookup("java:comp/env/jdbc/titanDB");
            return ds.getConnection();
        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }
```

Методы `byCheck()` и `byCredit()` содержат некоторые действия по проверке правильности данных перед их обработкой. Метод `byCredit()` проверяет, не истек ли срок годности кредитной карточки, сравнивая его с текущей датой. Если это происходит, возбуждается `PaymentException`.

Метод `byCheck()` сравнивает серийный номер чека с некоторым минимумом, который определен свойством, заданным во время развертывания компонента. Если проверяемое число меньше этого значения, возбуждается исключение `PaymentException`. Свойство для проверки возвращается методом `getMinCheckNumber()`. Для чтения значения свойства `minCheckNumber` мы можем использовать JNDI ENC:

```
    private int getMinCheckNumber() {
        try {
            InitialContext jndiCntx = new InitialContext();
            Integer value = (Integer)
                jndiCntx.lookup("java:comp/env/minCheckNumber");
            return value.intValue();
```

```
        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }
```

В данном случае для изменения прикладного поведения компонента мы используем набор свойств окружения из дескриптора развертывания. Очень удобно хранить пороги и другие ограничения в свойствах среды компонента, а не жестко кодировать их в программе. Это сообщает системе большую гибкость. Если, например, в системе «Титан» будет увеличен минимальный номер чека, то потребуется изменить лишь дескриптор развертывания компонента, а не определение класса. (Можно организовать получение этой информации непосредственно из базы данных.)

JNDI ENC: Доступ к свойствам окружения

В EJB соглашения между контейнером и компонентом включают контекст имен окружения JNDI (JNDI ENC). JNDI ENC – это пространство имен JNDI, специфичное для каждого типа компонента. На это пространство имен можно ссылаться по имени `"java:comp/env"` из любого компонента, а не только из объектных компонентов. Этот контекст имен предоставляет гибкий, но вместе с тем стандартный механизм для доступа к свойствам, другим компонентам и ресурсам контейнера.

Мы уже неоднократно сталкивались с JNDI ENC. В главе 10 он применялся для доступа к фабрике ресурсов `DataSource`. Компонент `ProcessPaymentBean` также использует JNDI ENC в методе `getConnection()` для доступа к `DataSource`. Кроме этого, он использует JNDI ENC в `getMinCheckNumber()` – методе для доступа к свойству окружения. В этом разделе рассматривается применение JNDI ENC для обращения к свойствам окружения.

В дескрипторе развертывания компонента могут быть объявлены именованные свойства (named properties). Компонент во время выполнения обращается к этим свойствам с помощью JNDI ENC. Эти свойства могут иметь тип `String` или один из примитивных типов-оболочек: `Integer`, `Long`, `Double`, `Float`, `Byte`, `Boolean` или `Short`. Модифицируя дескриптор развертывания, можно изменять поведение компонента во время развертывания без изменения его кода. Так, в компоненте ProcessPayment мы могли изменить минимальный номер принимаемого чека, меняя при развертывании свойство `minCheckNumber`. Два компонента ProcessPayment, развернутые в разных контейнерах, могут иметь разные минимальные номера чеков.

Здесь показано, как объявить именованное свойство:

```
<ejb-jar>
    <enterprise-beans>
```

```
        <session>
            <env-entry>
                <env-entry-name>minCheckNumber</env-entry-name>
                <env-entry-type>java.lang.Integer</env-entry-type>
                <env-entry-value>2000</env-entry-value>
            </env-entry>
            ...
        </session>
        ...
    </enterprise-beans>
    ...
</ejb-jar>
```

EJBContext

В EJB 2.0 и 1.1 метод `Context.getEnvironment()` не обязателен, т. е. он может не поддерживаться. Если он не поддерживается, то при его вызове будет сгенерировано исключение `RuntimeException`. Если он поддерживается, то возвратит только те значения, которые объявлены в дескрипторе развертывания следующим образом (здесь `minCheckNumber` — это имя свойства):

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <env-entry>
                <env-entry-name>ejb10-properties/minCheckNumber</env-entry-
name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>20000</env-entry-value>
            </env-entry>
            ...
        </session>
        ...
    </enterprise-beans>
    ...
</ejb-jar>
```

Подмножество `ejb10-properties` указывает, что свойство `minCheckNumber` доступно и из JNDI ENC `"java:comp/env/ejb10-properties/minCheckNumber"` (в виде значения `String`), и из метода `getEnvironment()`.

Только те свойства, которые объявлены внутри подмножества `ejb10-properties`, доступны через `EJBContext`. Кроме того, эти свойства доступны через `EJBContext` только в тех контейнерах, которые поддерживают метод `getEnvironment()` из EJB 1.0, все остальные контейнеры генерируют исключение `RuntimeException`. Предполагается, что большинство производителей EJB 2.0 откажутся от поддержки этой особенности, а разработчики для получения значения свойства будут использовать JNDI ENC вместо метода `EJBContext.getEnvironment()`.

Дескриптор развертывания компонента ProcessPayment

Развертывание компонента ProcessPayment не представляет никаких особенных сложностей. Оно, по существу, ничем не отличается от развертывания объектных компонентов, за исключением того, что у компонента ProcessPayment нет ни первичного ключа, ни полей постоянства. Ниже приведен XML-дескриптор развертывания для компонента ProcessPayment:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <session>
            <description>
                Служба, поддерживающая оплату.
            </description>
            <ejb-name>ProcessPaymentEJB</ejb-name>
            <home>
                com.titan.processpayment.ProcessPaymentHomeRemote
            </home>
            <remote>
                com.titan.processpayment.ProcessPaymentRemote
            </remote>
            <ejb-class>
                com.titan.processpayment.ProcessPaymentBean
            </ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <env-entry-name>minCheckNumber</env-entry-name>
                <env-entry-type>java.lang.Integer</env-entry-type>
                <env-entry-value>2000</env-entry-value>
            </env-entry>
            <resource-ref>
                <description>DataSource for the Titan database</description>
                <res-ref-name>jdbc/titanDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </session>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>
                This role represents everyone who is allowed full access
                to the ProcessPayment EJB.
            </description>
            <role-name>everyone</role-name>
```

```
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>ProcessPaymentEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>ProcessPaymentEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Дескриптор развертывания для EJB 1.1 имеет точно такой же вид, за исключением того, что его заголовок указывает спецификацию EJB 1.1 и дескриптор развертывания:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

📖 Рабочее упражнение 12.1. Сеансовый компонент без состояния

EJB 2.0: Локальные компонентные интерфейсы

Как и объектные компоненты, сеансовые компоненты без состояния могут определять локальные компонентные интерфейсы. Благодаря этому локальные интерфейсы сеансового компонента без состояния могут использоваться другими совмещенными компонентами, включая другие сеансовые компоненты без состояния и с состоянием и даже объектными компонентами. Очевидно, что применение локальных компонентных интерфейсов между двумя компонентами, расположенными в одной контейнерной системе, эффективнее, чем работа с удаленными интерфейсами.

Процесс создания локальных интерфейсов для сеансового компонента без состояния или с состоянием такой же, как и для объектных компонентов. Локальные интерфейсы расширяют `javax.ejb.EJBLocalObject` (для прикладных методов) и `javax.ejb.EJBLocalHome` (для внутренних интерфейсов). Затем эти интерфейсы определяются в XML-дескрипторе развертывания в элементах `<local>` и `<local-home>`.

Для краткости мы не будем определять локальные интерфейсы ни для компонента без состояния ProcessPayment, ни для компонента с состоянием TravelAgent, разработка которого обсуждается позже в этой главе. Опыт создания локальных интерфейсов для объектных компо-

нентов, который вы получили в главах 5, 6 и 7, может быть с легкостью применен к любому типу сеансовых компонентов.

Жизненный цикл сеансового компонента без состояния

Как и у объектного компонента, у сеансового компонента без состояния есть четко обозначенный жизненный цикл. Жизненный цикл сеансового компонента без состояния имеет два состояния: *не существует* (*Does Not Exist*) и *пул готовых методов* (*Method-Ready Pool*). Пул готовых методов похож на пул экземпляров, используемый для объектных компонентов. Он является одним из существенных различий между жизненными циклами сеансовых компонентов без состояния и с состоянием. В жизненном цикле компонентов без состояния определен пул экземпляров, а для компонентов с состоянием – нет.[1] На рис. 12.1 показаны состояния и переходы, через которые экземпляр сеансового компонента без состояния проходит в течение времени своей жизни.

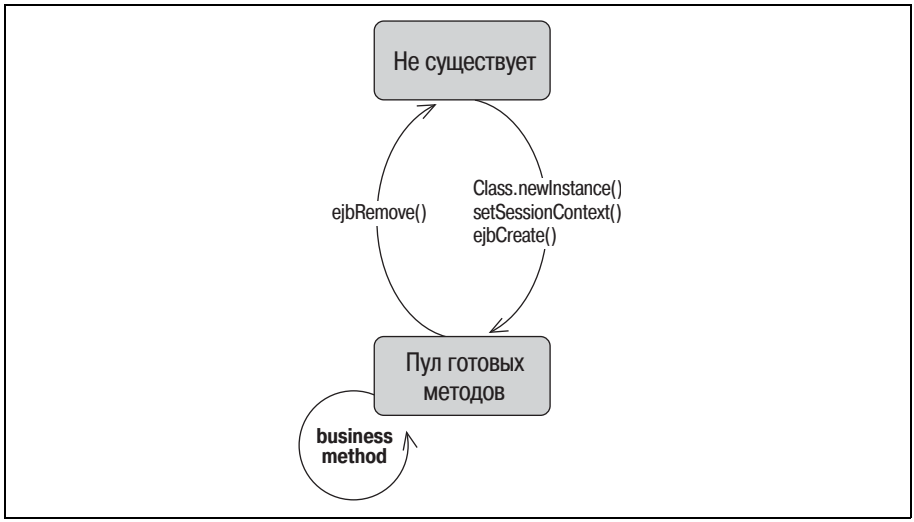

Рис. 12.1. Жизненный цикл сеансового компонента без состояния

[1] Некоторые производители не используют пул экземпляров без состояния, а могут вместо этого создавать и уничтожать экземпляры при каждом вызове метода. Это решение относится к деталям конкретной реализации и не влияет на установленный жизненный цикл экземпляра компонента без состояния.

Не существует

Когда компонент находится в состоянии «не существует», он не является экземпляром в памяти системы. Другими словами, он еще не создан.

Пул готовых методов

Экземпляры компонента без состояния перемещаются в пул готовых методов, как только они становятся нужны контейнеру. Когда сервер EJB запускается в первый раз, он может создать несколько экземпляров компонентов без состояния и поместить их в пул готовых методов. (Действительное поведение сервера зависит от его реализации.) Когда количество экземпляров без состояния, обслуживающих клиентские запросы, становится недостаточным, могут быть созданы и добавлены в пул дополнительные экземпляры.

Переход в пул готовых методов

При переходе экземпляра из состояния «не существует» в пул готовых методов выполняются три действия. Сначала с помощью вызова метода `Class.newInstance()` класса компонента без состояния создается экземпляр компонента.

Затем вызывается метод `SessionBean.setSessionContext(SessionContext context)` этого экземпляра компонента. В этом месте экземпляр принимает ссылку на свой `EJBContext`, служащий ему в течение всей его жизни. Ссылка `SessionContext` может быть сохранена в постоянном поле экземпляра сеансового компонента без состояния.

И наконец, для экземпляра компонента вызывается метод без аргументов `ejbCreate()`. Напомним, что сеансовый компонент без состояния содержит только один метод `ejbCreate()`, не принимающий ни одного параметра. Метод `ejbCreate()` вызывается только один раз за весь жизненный цикл сеансового компонента без состояния. Когда клиент вызывает метод `create()` внутреннего объекта, вызов не перенаправляется экземпляру компонента.

Объектные компоненты, сеансовые и управляемые сообщениями компоненты никогда не должны определять конструкторы. Позаботьтесь о необходимой инициализации внутри `ejbCreate()` и других методов обратного вызова. Контейнер создает экземпляр класса компонента с помощью метода `Class.newInstance()`, который требует наличия конструктора без аргументов.

Сеансовые компоненты без состояния не являются объектами, подлежащими активации, поэтому они могут поддерживать открытые под-

ключения к ресурсам в течение всего своего жизненного цикла.[1] Метод `ejbRemove()` должен закрывать все открытые ресурсы, прежде чем сеансовый компонент без состояния в конце своего жизненного цикла будет удален из памяти. Метод `ejbRemove()` рассматривается более подробно далее в этом разделе.

Жизнь в пуле готовых методов

Если экземпляр находится в пуле готовых методов, он готов обслуживать клиентские запросы. Когда клиент вызывает прикладной метод компонентного объекта, данный запрос метода перенаправляется любому доступному экземпляру, находящемуся в пуле готовых методов. Во время выполнения запроса экземпляр недоступен для других компонентных объектов. Как только экземпляр закончит выполнение запроса, он немедленно станет доступным любому компонентному объекту, которому он будет нужен. Все это немного отличается от пула экземпляров для объектных компонентов, описанного в главе 11. В пуле экземпляров объектных компонентов экземпляр компонента может переключаться компонентным объектом для обслуживания нескольких вызовов методов. Сеансовые экземпляры без состояния, однако, как правило, выделяются для компонентного объекта только на время одного вызова метода.

Хотя разные производители могут применять разные стратегии поддержки сеансовых компонентов без состояния, вероятнее всего они будут использовать стратегию подкачки экземпляров, подобную той, которая применяется для объектных компонентов (глава 11). Однако обмен очень кратковременен, его длительность равна времени выполнения прикладного метода. Когда экземпляр переключается, происходит изменение его контекста `SessionContext` для того, чтобы отразить контекст его компонентного объекта и клиента, вызвавшего метод. Экземпляр компонента может быть включен в контекст транзакции клиентского запроса и может получить доступ к информации в `SessionContext`, специфичной для данного клиентского запроса: например, безопасность и транзакционные методы. После того как экземпляр закончит обслуживать клиента, он отсоединяется от компонентного объекта и возвращается в пул готовых методов.

Сеансовые компоненты без состояния не являются объектами, подлежащими активации, и никогда не обрабатывают вызовы своих методов обратного вызова `ejbActivate()` и `ejbPassivate()`. Причина проста: экземпляры без состояния не поддерживают какое-либо состояние

[1] Предполагается, что продолжительность жизни экземпляра компонента без состояния очень велика. Однако в действительности некоторые серверы EJB могут уничтожать и создавать экземпляры при каждом вызове метода, делая такую тактику менее привлекательной. Детали обработки экземпляров без состояния описываются в документации по серверу.

диалога, которое нужно было бы сохранять. (Как мы увидим позже, сеансовые компоненты *с состоянием* зависят от активации.)

Клиенты, которым требуется удаленная или локальная ссылка на сеансовый компонент без состояния, начинают с вызова метода `create()` внутреннего объекта компонента:

```
Object ref = jndiConnection.lookup("ProcessPaymentHomeRemote");
ProcessPaymentHomeRemote home = (ProcessPaymentHomeRemote)
    PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);

ProcessPaymentRemote pp = home.create();
```

В отличие от объектного компонента и сеансового компонента с состоянием, вызов метода `create()` не приводит к вызову метода `ejbCreate()` компонента. В сеансовых компонентах без состояния вызов метода `create()` внутреннего объекта приводит к созданию для клиента компонентного объекта (и только).

Метод `ejbCreate()` сеансового компонента без состояния вызывается лишь один раз за весь жизненный цикл экземпляра, в момент его перехода из состояния «не существует» в пул готовых методов. Он не вызывается повторно каждый раз, когда клиент запрашивает удаленную ссылку на компонент. Сеансовые компоненты без состояния ограничены единственным методом без аргументов `create()`, т. к. контейнер не имеет никакой возможности определить, какой метод `create()` вызовет клиент.

Выход из пула готовых методов: Смерть экземпляра компонента без состояния

Экземпляры компонентов покидают пул готовых методов и переходят в состояние «не существует», когда они больше не нужны серверу, т. е. когда сервер решает уменьшить общий размер пула готовых методов, удаляя один или несколько экземпляров из памяти. Этот процесс начинается с вызова метода `ejbRemove()` экземпляра. В это время экземпляр компонента должен выполнить необходимые завершающие действия, такие как закрытие открытых ресурсов. Метод `ejbRemove()` вызывается только один раз за весь жизненный цикл экземпляра сеансового компонента без состояния – перед тем, как он будет переведен в состояние «не существует». Когда клиент вызывает один из удаляющих методов удаленного или внутреннего интерфейса сеансового компонента без состояния, этот вызов не передается экземпляру компонента. Клиентский вызов метода просто делает заглушку недействительной и удаляет компонентный объект: это уведомляет контейнер о том, что данный компонент больше не нужен клиенту. Контейнер сам вызывает метод `ejbRemove()` экземпляра без состояния, но только в конце его жизненного цикла. И это тоже отличается и от сеансовых компонентов с состоянием, и от объектных компонентов, где вызов клиентом удаляющего метода приводит к более разрушительным послед-

ствиям. Во время выполнения метода `ejbRemove()` доступ экземпляра компонента к `SessionContext` и JNDI ENC все еще возможен. После завершения метода `ejbRemove()` ссылка на компонент удаляется, и через какое-то время он собирается сборщиком мусора.

Сеансовый компонент с состоянием

Каждый сеансовый компонент с состоянием выделяется одному клиенту на все время жизни экземпляра компонента; он действует в качестве посредника от имени своего клиента. В отличие от сеансовых компонентов без состояния и объектных компонентов, сеансовые компоненты с состоянием не переключаются между компонентными объектами и не сохраняются в пуле экземпляров. После того как сеансовый компонент с состоянием создан и связан с компонентным объектом, он выделяется этому компонентному объекту на время всего своего жизненного цикла.[1]

Сеансовые компоненты с состоянием поддерживают состояние диалога, это означает, что переменные экземпляра класса компонента могут кэшировать данные, имеющие отношение к клиенту, в промежутках между вызовами методов. Это делает возможной взаимозависимость методов, при которой изменения состояния компонента, сделанные в одном вызове метода, могут влиять на результаты последующих вызовов. В противоположность этому сеансовые компоненты без состояния, которые мы только что рассмотрели, не поддерживают состояние диалога. Хотя компоненты без состояния могут иметь переменные экземпляра, эти поля не относятся к конкретному клиенту. Экземпляр без состояния переключается между несколькими компонентными объектами, поэтому нельзя предсказать, какой экземпляр будет обслуживать вызов метода. В случае сеансовых компонентов с состоянием каждый вызов метода от клиента обслуживается одним и тем же экземпляром (по крайней мере, концептуально), поэтому состояние экземпляра компонента от одного вызова метода до другого можно предсказать.

Хотя сеансовые компоненты с состоянием поддерживают состояние диалога, сами они, в отличие от объектных компонентов, не являются постоянными. Объектные компоненты представляют данные в базе данных; их поля постоянства переписываются непосредственно в базу данных. Сеансовые компоненты с состоянием, как и компоненты без состояния, могут обращаться к базе данных, но не представляют в ней

[1] Это концептуальная модель. В действительности некоторые контейнеры EJB могут использовать для сеансовых компонентов с состоянием подкачку экземпляров, создавая видимость обслуживания всех запросов одним экземпляром. Однако концептуально один экземпляр сеансового компонента с состоянием обслуживает все запросы.

никаких данных. Кроме этого, компоненты с состоянием не используются одновременно как объектные компоненты. Если имеется компонентный объект, представляющий экземпляр судна с названием «Paradise», то все клиентские запросы для этого судна будут координироваться этим же компонентным объектом.[1] В случае сеансовых компонентов с состоянием компонентный объект выделяется для одного клиента, следовательно, сеансовые компоненты с состоянием не используются одновременно.

Сеансовые компоненты с состоянием часто рассматриваются как расширение клиента. Это имеет смысл, если представить клиент как набор операций и состояний. Каждая задача может полагаться на некоторую информацию, полученную или модифицированную предыдущей операцией. Прекрасный пример этому GUI-клиент: когда мы заполняем поля графического пользовательского интерфейса (Graphic User Interface, GUI), мы тем самым создаем состояние диалога. Нажатие кнопки запускает операцию заполнения нескольких полей на основе информации, только что введенной нами. Информация в полях отражает состояние диалога.

Сеансовые компоненты с состоянием позволяют инкапсулировать некоторую часть прикладной логики и состояние диалога с клиентом и переместить ее на сервер. Размещение этой логики на сервере облегчает клиентское приложение и в целом делает управление системой проще. Сеансовый компонент с состоянием действует как посредник клиента, который с целью выполнения набора задач управляет процессами или *рабочим потоком*. Он управляет взаимодействиями между другими компонентами и, кроме того, для выполнения сложных задач, управляет непосредственным доступом к данным во множественных операциях. Инкапсулируя логику и управляя рабочим потоком от имени клиента, компоненты с состоянием представляют простой интерфейс, скрывающий от клиента детали многих взаимосвязанных операций над базой данных и другими компонентами.

EJB 2.0: Подготовка к использованию TravelAgent

Компонент TravelAgent будет работать с компонентами Cabin, Cruise, Reservation и Customer, разработанными в главах 6 и 7. Он будет координировать взаимодействия между этими объектными компонентами при внесении пассажира в список круиза.

Компонент Reservation, который рассматривался в главе 7, будет немного изменен так, чтобы он мог создаваться сразу же с заданными

[1] Это тоже концептуальная модель. Некоторые контейнеры могут разделять компонентные объекты для параллельного доступа к одному компоненту, рассчитывая на то, что управление параллелизмом возьмет на себя база данных. Однако концептуально конечный результат такой же.

полями отношений. Для того чтобы сделать это, выполним перегрузку его метода `ejbCreate()`:

```
public abstract class ReservationBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(CustomerRemote customer, CruiseLocal cruise,
        CabinLocal cabin, double price, Date dateBooked) {

        setAmountPaid(price);
        setDate(dateBooked);
        return null;
    }
    public void ejbPostCreate(CustomerRemote customer, CruiseLocal cruise,
        CabinLocal cabin, double price, Date dateBooked)
        throws javax.ejb.CreateException {

        setCruise(cruise);

        // добавляем Cabin в поле-коллекцию CMR
        Set cabins = new HashSet();
        cabins.add(cabin);
        this.setCabins(cabins);

        try {
            Integer primKey = (Integer)customer.getPrimaryKey();
            javax.naming.Context jndiContext = new InitialContext();
            CustomerHomeLocal home = (CustomerHomeLocal)
                jndiContext.lookup("java:comp/env/ejb/CustomerHomeLocal");
            CustomerLocal custL = home.findByPrimaryKey(primKey);

            // добавляем Customer в поле-коллекцию CMR
            Set customers = new HashSet();
            customers.add(custL);
            this.setCustomers(customers);

        } catch (RemoteException re) {
            throw new CreateException("Invalid Customer");
        } catch (FinderException fe) {
            throw new CreateException("Invalid Customer");
        } catch (NamingException ne) {
            throw new CreateException("Invalid Customer");
        }
    }
```

Поля отношений используют локальную ссылку на компонентный объект, поэтому для того, чтобы установить поле отношения `customer` компонента Reservation, мы должны конвертировать ссылку на `CustomerRemote` в ссылку на `CustomerLocal`. Чтобы сделать это, вы можете либо воспользоваться JNDI ENC для поиска локального внутреннего интерфейса, а затем выполнить его метод `findByPrimaryKey()`, либо реализовать в компоненте Reservation метод `ejbSelect()`, ищущий ссылку на `CustomerLocal`.

EJB 1.1: Подготовка к использованию TravelAgent

И компонент TravelAgent, и компонент ProcessPayment, который мы создаем в этой главе, зависят от других объектных компонентов, часть из которых мы разработали ранее в этой книге, а остальные можно загрузить с веб-сайта O'Reilly. В главе 4 мы создали компонент Cabin, но для нашего примера нам понадобятся еще несколько компонентов, а именно: компоненты Cruise, Customer и Reservation. Исходный текст этих компонентов вместе с остальными примерами этой книги доступен на сайте O'Reilly. Инструкции для загрузки кода приводятся во введении к данной книге и в рабочих книгах.

Для того чтобы использовать эти компоненты, необходимо создать в своей базе данных несколько новых таблиц. Здесь приведены определения для таблиц, которые понадобятся для новых объектных компонентов. Компонент Cruise сопоставлен с таблицей `CRUISE`:

```
CREATE TABLE CRUISE
(
    ID          INT PRIMARY KEY,
    NAME        CHAR(30),
    SHIP_ID     INT
)
```

Компонент Customer сопоставлен с таблицей `CUSTOMER`:

```
CREATE TABLE CUSTOMER
(
    ID              INT PRIMARY KEY,
    FIRST_NAME      CHAR(30),
    LAST_NAME       CHAR(30),
    MIDDLE_NAME     CHAR(30)
)
```

Компонент Reservation сопоставлен с таблицей `RESERVATION`:

```
CREATE TABLE RESERVATION
(
    CUSTOMER_ID     INT,
    CABIN_ID        INT,
    CRUISE_ID       INT,
    AMOUNT_PAID DECIMAL (8,2),
    DATE_RESERVED DATE
)
```

Создав эти таблицы, приступайте к развертыванию на сервере EJB данных компонентов в виде управляемых контейнером объектов и протестируйте их, чтобы убедиться, что они работают нормально.

Компонент TravelAgent

Компонент TravelAgent, с которым мы уже встречались, представляет собой сеансовый компонент с состоянием, содержащий процесс заказа билетов на круиз. В данной главе мы будем дальше разрабатывать этот компонент, чтобы показать, как могут применяться сеансовые компоненты с состоянием в качестве объектов рабочего потока.

Хотя читатели, работающие с EJB 2.0, будут использовать для других компонентов локальные интерфейсы, мы не будем разрабатывать локальный интерфейс для компонента TravelAgent. Правила разработки локальных интерфейсов сеансовых компонентов с состоянием такие же, как и для сеансовых компонентов без состояния и объектных компонентов. Компонент TravelAgent предназначен для использования только удаленными клиентами и поэтому не требует набора локальных компонентных интерфейсов.

TravelAgent: Удаленный интерфейс

В главе 4 мы разработали первую версию интерфейса `TravelAgentRemote`, который содержал единственный прикладной метод `listCabins()`. Сейчас мы удалим метод `listCabins()` и переопределим компонент TravelAgent так, чтобы он вел себя как объект рабочего потока. Далее в этой главе мы добавим модифицированный метод, чтобы дать возможность пользователю получать более определенный список кают.

Как сеансовый компонент с состоянием, моделирующий рабочий поток, компонент TravelAgent управляет взаимодействиями между несколькими компонентами и одновременно поддерживает состояние диалога. Следующий код содержит модифицированный интерфейс `TravelAgentRemote`:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
import com.titan.processpayment.CreditCardDO;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(Integer cruise)
        throws RemoteException, FinderException;

    public void setCabinID(Integer cabin)
        throws RemoteException, FinderException;

    public TicketDO bookPassage(CreditCardDO card, double price)
        throws RemoteException,IncompleteConversationalState;
}
```

Назначение компонента TravelAgent состоит в том, чтобы делать заказы билетов для круиза. Для выполнения этой задачи компонент должен знать, какие круиз, каюта и пассажир составляют данный заказ. Поэтому клиент, использующий компонент TravelAgent, должен перед созданием заказа собрать эту информацию. Интерфейс `TravelAgentRemote` предоставляет методы для установки идентификаторов круиза и каюты, которые клиент хочет заказать. Можно предположить, что идентификатор каюты берется из списка, а идентификатор круиза – из какого-то другого источника. Пассажир определяется в методе `create()` внутреннего интерфейса – подробнее это рассматривается позже.

После того как выбраны пассажир, круиз и каюта, компонент TravelAgent готов к обработке заказа. Эта операция выполняется в методе `bookPassage()`, которому требуется информация о кредитной карточке пассажира и цене круиза. `bookPassage()` отвечает за прием платежей со счета пассажира, резервирование выбранной каюты на нужном судне нужного круиза и подготовку билета для пассажира. Как это делается, сейчас нам не важно, поскольку, разрабатывая удаленный интерфейс, мы имеем дело лишь с прикладным определением компонента. Мы рассмотрим реализацию, когда будем говорить о классе компонента.

Обратите внимание, что метод `bookPassage()` генерирует специфичное для приложения исключение `IncompleteConversationalState`. Это исключение предназначено для указания на прикладную ошибку, которая может произойти при регистрации пассажира на круиз. Исключение `IncompleteConversationalState` указывает, что у компонента TravelAgent нет всей необходимой информации, требуемой для обработки заказа. Класс прикладного исключения `IncompleteConversationalState` определен следующим образом:

```
package com.titan.travelagent;

public class IncompleteConversationalState extends java.lang.Exception {
    public IncompleteConversationalState(){super();}
    public IncompleteConversationalState(String msg){super(msg);}
}
```

Зависимый объект: TicketDO

Так же как компонент ProcessPayment использует классы `CreditCardDO` и `CheckDO`, класс `TicketDO`, возвращаемый методом `bookPassage()`, определен как объект, передаваемый по значению. Можно было бы возразить, что билет (ticket) должен являться объектным компонентом, поскольку он достаточно независим и мог бы быть доступен вне контекста компонента TravelAgent. Однако решение о способе использования прикладного объекта также может определять, должен ли он быть компонентом или просто классом.

Объект TicketDO, например, может быть защищен цифровой подписью и отправлен заказчику по электронной почте в качестве подтверждения оплаты. Это невозможно было бы сделать, если бы объект TicketDO был определен в виде объектного компонента. На компоненты можно ссылаться только через их компонентные интерфейсы и их нельзя передавать по значению, как сериализуемые объекты, такие как TicketDO, CreditCardDO и CheckDO. В качестве упражнения по передаче по значению мы определим TicketDO в виде простого сериализуемого объекта, а не компонента.

EJB 2.0 использует локальные интерфейсы компонентов Cruise и Cabin и удаленный интерфейс компонента Customer при создании нового объекта TicketDO:

```
package com.titan.travelagent;

import com.titan.cruise.CruiseLocal;
import com.titan.cabin.CabinLocal;
import com.titan.customer.CustomerRemote;

public class TicketDO implements java.io.Serializable {
    public Integer customerID;
    public Integer cruiseID;
    public Integer cabinID;
    public double price;
    public String description;

    public TicketDO(CustomerRemote customer, CruiseLocal cruise,
        CabinLocal cabin, double price) throws javax.ejb.FinderException,
        RemoteException, javax.naming.NamingException {

            description = customer.getFirstName()+
                " " + customer.getLastName() +
                " has been booked for the "
                + cruise.getName() +
                " cruise on ship " +
                  cruise.getShip().getName() + ".\n" +
                " Your accommodations include " +
                  cabin.getName() +
                " a " + cabin.getBedCount() +
                " bed cabin on deck level " + cabin.getDeckLevel() +
                ".\n Total charge = " + price;
            customerID = (Integer)customer.getPrimaryKey();
            cruiseID = (Integer)cruise.getPrimaryKey();
            cabinID = (Integer)cabin.getPrimaryKey();
            this.price = price;
        }

    public String toString() {
        return description;
    }
}
```

EJB 1.1 также использует удаленные интерфейсы компонентов Customer, Cruise и Cabin при создании нового объекта TicketDO:

```
package com.titan.travelagent;

import com.titan.cruise.CruiseRemote;
import com.titan.cabin.CabinRemote;
import com.titan.customer.CustomerRemote;
import java.rmi.RemoteException;

public class TicketDO implements java.io.Serializable {
    public Integer customerID;
    public Integer cruiseID;
    public Integer cabinID;
    public double price;
    public String description;

    public TicketDO(CustomerRemote customer, CruiseRemote cruise,
        CabinRemote cabin, double price) throws javax.ejb.FinderException,
        RemoteException, javax.naming.NamingException {

            description = customer.getFirstName()+
                " " + customer.getLastName() +
                " has been booked for the "
                + cruise.getName() +
                " cruise on ship " + cruise.getShipID() + ".\n" +
                " Your accommodations include " +
                  cabin.getName() +
                " a " + cabin.getBedCount() +
                " bed cabin on deck level " + cabin.getDeckLevel() +
                ".\n Total charge = " + price;
            customerID = (Integer)customer.getPrimaryKey();
            cruiseID = (Integer)cruise.getPrimaryKey();
            cabinID = (Integer)cabin.getPrimaryKey();
            this.price = price;

        }
    public String toString() {
        return description;
    }
}
```

TravelAgentHomeRemote: Внутренний интерфейс

Взяв за основу интерфейс TravelAgentHomeRemote, разработанный нами в главе 4, мы можем изменить метод create() так, чтобы он принимал удаленную ссылку на пассажира, делающего заказ билетов:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import com.titan.customer.CustomerRemote;
```

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {
    public TravelAgentRemote create(CustomerRemote cust)
        throws RemoteException, CreateException;
}
```

Метод `create()` этого внутреннего интерфейса требует, чтобы удаленная ссылка на компонент Customer применялась для создания компонента TravelAgent. Ввиду отсутствия других методов `create()` невозможно создать компонент TravelAgent, если неизвестно, кто является пассажиром. Ссылка на компонент Customer предоставляет компоненту TravelAgent некоторую часть состояния диалога, которую необходимо обработать в методе `bookPassage()`.

Взгляд со стороны клиента

Прежде чем окончательно завершить определения интерфейсов для нашего компонента, неплохо было бы представить себе, как компонент будет применяться клиентами. Представьте, что компонент TravelAgent используется Java-приложением с GUI-полями. GUI-поля служат для ввода круиза и типа каюты, которые предпочитает пассажир. Начнем с рассмотрения кода, используемого в начале процесса резервирования:

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("CustomerHomeRemote");
CustomerHomeRemote customerHome =(CustomerHomeRemote)
    PortableRemoteObject.narrow(ref, CustomerHomeRemote.class);

String ln = tfLastName.getText();
String fn = tfFirstName.getText();
String mn = tfMiddleName.getText();
CustomerRemote customer = customerHome.create(nextID, ln, fn, mn);

ref = jndiContext.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote home = (TravelAgentHomeRemote)
    PortableRemoteObject.narrow(ref, TravelAgentHomeRemote.class);

TravelAgentRemote agent = home.create(customer);
```

В этом фрагменте кода на основе информации, собранной туристическим агентом по телефону, создается новый компонент Customer. Затем ссылка на `CustomerRemote` применяется для создания компонента TravelAgent. После этого из другой части приложения мы забираем выбранные круиз и каюту:

```
Integer cruise_id = new Integer(textField_cruiseNumber.getText());

Integer cabin_id = new Integer( textField_cabinNumber.getText());

agent.setCruiseID(cruise_id);
agent.setCabinID(cabin_id);
```

Туристический агент выбирает круиз и каюту, которые клиент желает зарезервировать. Эти идентификаторы устанавливаются в компоненте TravelAgent, который поддерживает состояние диалога всего процесса.

В конце процесса туристический агент завершает резервирование, обработав заказ и сгенерировав билет. Поскольку компонент TravelAgent поддерживает состояние диалога, кэшируя информацию о пассажире, каюте и круизе, для завершения транзакции ему необходимы только данные о кредитной карточке и стоимости:

```
String cardNumber = textField_cardNumber.getText();
Date date = dateFormatter.parse(textField_cardExpiration.getText());
String cardBrand = textField_cardBrand.getText();
CreditCardDO card = new CreditCardDO(cardNumber,date,cardBrand);
double price =
double.valueOf(textField_cruisePrice.getText()).doubleValue();
TicketDO ticket = agent.bookPassage(card,price);
PrintingService.print(ticket);
```

Теперь мы можем продолжить разработку. Приведенный обзор способов использования компонента TravelAgent клиентом подтверждает работоспособность наших определений удаленного и внутреннего интерфейсов.

TravelAgentBean: Класс компонента

Теперь мы можем полностью реализовать поведение компонента TravelAgent, заданное новым удаленным и внутренним интерфейсами.[1]

Здесь приведена часть определения нового класса `TravelAgentBean` для EJB 2.0:

```
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;
import com.titan.processpayment.*;
import com.titan.cruise.*;
import com.titan.customer.*;
import com.titan.cabin.*;

public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
```

[1] Модифицируя компонент, разработанный в главе 4, не забудьте удалить метод `listCabin()`. Далее в этой главе мы добавим новую реализацию этого метода.

```
public CruiseLocal cruise;
public CabinLocal cabin;

public javax.ejb.SessionContext ejbContext;

public javax.naming.Context jndiContext;

public void ejbCreate(CustomerRemote cust) {
    customer = cust;
}
public void setCabinID(Integer cabinID) throws javax.ejb.FinderException
{
    try {
        CabinHomeLocal home = (CabinHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");

        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID) throws
  javax.ejb.FinderException {
    try {
       CruiseHomeLocal home = (CruiseHomeLocal)
           jndiContext.lookup("java:comp/env/ejb/CruiseHomeLocal");

       cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }

}
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null)
    {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
           jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price, new Date());

        Object ref = jndiContext.lookup("java:comp/env/ejb/
            ProcessPaymentHomeRemote");

        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow (ref, ProcessPaymentHomeRe-
                                         mote.class);
```

```
            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer, cruise, cabin, price);
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void setSessionContext(javax.ejb.SessionContext cntx)
    {

        ejbContext = cntx;
        try {
            jndiContext = new javax.naming.InitialContext();
        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }
}
```

В EJB 1.1 определение класса `TravelAgentBean` **будет выглядеть так:**

```
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;
import com.titan.processpayment.*;
import com.titan.cruise.*;
import com.titan.customer.*;
import com.titan.cabin.*;

public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
    public CruiseRemote cruise;
    public CabinRemote cabin;

    public javax.ejb.SessionContext ejbContext;

    public javax.naming.Context jndiContext;

    public void ejbCreate(CustomerRemote cust) {
        customer = cust;
    }
```

```
public void setCabinID(Integer cabinID) throws javax.ejb.FinderException
{
    try {
        CabinHomeRemote home = (CabinHomeRemote)
            getHome("CabinHomeRemote", CabinHomeRemote.class);
        cabin = home.findByPrimaryKey(cabinID);
    } catch(Exception re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID) throws
  javax.ejb.FinderException {
    try {
        CruiseHomeRemote home = (CruiseHomeRemote)
            getHome("CruiseHomeRemote", CruiseHomeRemote. class);
        cruise = home.findByPrimaryKey(cruiseID);
    } catch(Exception re) {
        throw new EJBException(re);
    }

}
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null){
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome = (ReservationHomeRemote)
          getHome("ReservationHomeRemote", ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price, new Date());
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            getHome("ProcessPaymentHomeRemote", ProcessPaymentHomeRe-
                    mote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}

public void setSessionContext(javax.ejb.SessionContext cntx)
{

    ejbContext = cntx;
```

```
            try {
                jndiContext = new javax.naming.InitialContext();
            } catch(NamingException ne) {
                throw new EJBException(ne);
            }
        }
        protected Object getHome(String name,Class type) {
            try {
                Object ref = jndiContext.lookup("java:comp/env/ejb/"+name);
                return PortableRemoteObject.narrow(ref, type);
            } catch(NamingException ne) {
                throw new EJBException(ne);
            }
        }
    }
```

Определение класса `TravelAgentBean` содержит достаточно много кода, поэтому рассмотрим его по частям. Сначала посмотрим на метод `ejbCreate()`:

```
public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
        ...

    public javax.ejb.SessionContext ejbContext;
    public javax.naming.Context jndiContext;

    public void ejbCreate(CustomerRemote cust) {
        customer = cust;
    }
```

При создании компонента удаленная ссылка на компонент Customer передается экземпляру компонента и запоминается в поле `customer`. Поле `customer` является частью состояния диалога компонента. Мы могли бы идентифицировать клиента по целому числу ID и создать удаленную ссылку на компонент Customer в методе `ejbCreate()`. Однако мы передали непосредственно ссылку, чтобы показать, что удаленные компонентные ссылки могут передаваться компоненту из клиентского приложения. Они также могут возвращаться компонентом клиенту и передаваться между компонентами, расположенными как на одном, так и на разных серверах EJB.

Ссылки на `SessionContext` и контекст JNDI хранятся в полях с именами `ejbContext` и `jndiContext`. Приставки «ejb» и «jndi» позволяют избежать путаницы между контекстами разных типов.

Когда компонент пассивируется, JNDI ENC должен поддерживаться как часть состояния диалога компонента. Это означает, что контекст JNDI должен быть постоянным. После помещения ссылки на JNDI ENC в поле она остается действительной на протяжении всей жизни компонента. В `TravelAgentBean` в начале жизненного цикла компонента

при установке SessionContext мы задаем значение поля jndiContext так, чтобы оно ссылалось на JNDI ENC:

```
public void setSessionContext(javax.ejb.SessionContext cntx) {
    ejbContext = cntx;
    try {
        jndiContext = new InitialContext();
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

EJB-контейнер специальным образом обрабатывает ссылки на SessionContext, JNDI ENC, ссылки на другие компоненты (типы удаленных и внутренних интерфейсов) и на тип JTA UserTransaction, рассматриваемый подробно в главе 14. Контейнер должен поддерживать все поля экземпляров, которые ссылаются на объекты данных типов как часть состояния диалога, даже если они не сериализуемы. Все остальные поля при пассивации компонента должны быть либо сериализуемыми, либо нулевыми.

Компонент TravelAgent содержит методы для задания требуемых круиза и каюты. Эти методы принимают в качестве параметров идентификаторы типа Integer и возвращают из подходящего внутреннего интерфейса ссылки на подходящие компоненты Cruise и Cabin. Эти ссылки также являются частью состояния диалога компонента TravelAgent.

Здесь показано, как setCabinID() и getCabinID() используются в EJB 2.0:

```
public void setCabinID(Integer cabinID)
    throws javax.ejb.FinderException {
    try {
        CabinHomeLocal home = (CabinHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");

        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
        CruiseHomeLocal home = (CruiseHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/CruiseHomeLocal");

        cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
```

Далее показано, как `setCabinID()` и `getCabinID()` применяются в EJB 1.1:

```
public void setCabinID(Integer cabinID)
    throws javax.ejb.FinderException {
    try {
        CabinHomeRemote home = (CabinHome)
            getHome("CabinHomeRemote", CabinHome.class);
        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
        CruiseHome home = (CruiseHome)
            getHome("CruiseHomeRemote", CruiseHome.class);
        cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
```

Может показаться странным, что мы устанавливаем эти значения с помощью целочисленных идентификаторов, но сохраняем их в состоянии диалога в виде ссылок на объектные компоненты. Использование для данных объектов идентификаторов типа `Integer` проще для клиента, который не работает со ссылками на эти компоненты. В клиентском коде мы получаем идентификаторы каюты и круиза из текстовых полей. Для чего заставлять клиента получать ссылку на компоненты Cruise и Cabin, если идентификатор гораздо проще? Кроме того, с точки зрения сетевого трафика, применение идентификаторов дешевле, чем передача удаленной ссылки. В методе `bookPassage()` нам необходимы ссылки типа компонентного объекта на данный компонент, поэтому для получения ссылок на действительные объектные компоненты мы используем их идентификаторы. Перед восстановлением удаленных ссылок мы могли бы подождать вызова метода `bookPassage()`, но таким способом мы сохраняем простоту метода `bookPassage()`.

JNDI ENC и компонентные ссылки

Для получения ссылок на внутренние интерфейсы других компонентов можно использовать JNDI ENC. Это позволяет избежать жесткого кодирования в компоненте свойств JNDI, специфичных для конкретного производителя. Другими словами, JNDI ENC позволяет компонентным ссылкам распределяться по сети и быть независимыми от производителя.

В листинге для `TravelAgentBean` в EJB 2.0 мы использовали JNDI ENC для доступа и к удаленному внутреннему интерфейсу компонента Pro-

cessPayment, и к локальным внутренним интерфейсам компонентов Cruise и Cabin. Это иллюстрирует гибкость JNDI ENC, который может предоставлять каталоги и для локальных, и для удаленных компонентов.

В листинге для класса `TravelAgentBean` в EJB 1.1 `getHome()` представляет собой удобный метод для сокрытия деталей получения удаленных ссылок на внутренние объекты. Метод `getHome()` использует ссылку `jndiContext` для получения ссылок на внутренние объекты компонентов Cabin, Ship, ProcessPayment и Cruise.

Спецификация EJB рекомендует, чтобы все компонентные ссылки были привязаны к контексту `"java:comp/env/ejb"`, и здесь мы следуем этому соглашению. В компонент TravelAgent мы передаем имя нужного нам внутреннего объекта и для выполнения поиска добавляем его к контексту `"java:comp/env/ejb"`.

Удаленные компонентные ссылки в JNDI ENC. Для объявления удаленных компонентных ссылок дескриптор развертывания предоставляет специальный набор тегов. Здесь показано, как используются тег `<ejb-ref>` и его подэлементы:

```
<ejb-ref>
    <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>
        com.titan.processpayment.ProcessPaymentHomeRemote
    </home>
    <remote>
       com.titan.processpayment.ProcessPaymentRemote
    </remote>
</ejb-ref>
```

Смысл этих элементов должен быть понятен: они определяют имя компонента в области ENC, объявляют тип компонента и дают имена его удаленному и внутреннему интерфейсам. Во время развертывания компонента выполняется сопоставление элементов `<ejb-ref>` с действительными компонентами способом, специфичным для конкретного производителя. Элементы `<ejb-ref>` также могут быть связаны с компонентами сборщиком приложения в этом же самом развертывании (этот вопрос рассматривается подробно в главе 16). Разработчикам EJB 2.0 рекомендуется применять локальные компонентные интерфейсы компонентов, расположенных в том же приложении и контейнере.

Во время развертывания инструментальные средства контейнера EJB связывают удаленные ссылки, объявленные в элементах `<ejb-ref>`, с компонентами, которые могут быть расположены на той же машине или на другом сетевом узле.

EJB 2.0: локальные компонентные ссылки в JNDI ENC. Кроме этого, дескриптор развертывания предоставляет специальный набор тегов,

элементы <ejb-local-ref> для объявления локальных компонентных ссылок: компонентов, совмещенных в одном контейнере и развернутых в одном файле EJB JAR. Элементы <ejb-local-ref> следуют непосредственно за элементами <ejb-ref>:

```
<ejb-local-ref>
    <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>
        com.titan.cruise.CruiseHomeLocal
    </local-home>
    <local>
        com.titan.cruise.CruiseLocal
    </local>
    <ejb-link>CruiseEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
    <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>
        com.titan.cabin.CabinHomeLocal
    </local-home>
    <local>
        com.titan.cabin.CabinLocal
    </local>
    <ejb-link>CabinEJB</ejb-link>
</ejb-local-ref>
```

Элемент <ejb-local-ref> определяет имя компонента в области ENC, объявляет тип компонента и дает имена его локальным компонентным интерфейсам. Эти элементы лучше явно связать с другими совмещенными компонентами при помощи элемента <ejb-link>, хотя соединение их на этой стадии не обязательно – сборщик приложения или управляющий развертыванием может делать это позже. Значение элемента <ejb-link> внутри <ejb-local-ref> должно соответствовать значению <ejb-name> подходящего компонента в том же самом файле JAR.

Во время развертывания инструментальные средства контейнера EJB связывают локальные ссылки, объявленные в элементах <ejb-local-ref>, с объектными компонентами, совмещенными в одной контейнерной системе.

Метод bookPassage()

Последнее, что нас интересует в нашем определении компонента, – это метод bookPassage(). Он использует состояние диалога, накопленное методами ejbCreate(), setCabinID() и setCruiseID(), для обработки заказа билета пассажира.

Здесь показано, как метод bookPassage() применяется в EJB 2.0:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
```

```
        if (customer == null || cruise == null || cabin == null) {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeLocal resHome = (ReservationHomeLocal)
                jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price, new Date());

            Object ref = jndiContext.lookup("java:comp/env/ejb/
                ProcessPaymentHomeRemote");

            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
               PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
```

Здесь показано, как метод `bookPassage()` **используется в EJB 1.1:**

```
    public TicketDO bookPassage(CreditCardDO card, double price)
        throws IncompleteConversationalState {

        if (customer == null || cruise == null || cabin == null) {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeRemote resHome = (ReservationHomeRemote)
            getHome("ReservationHomeRemote", ReservationHomeRemote.class);
            ReservationRemote reservation =
                resHome.create(customer, cruise, cabin, price, new Date());
            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                getHome("ProcessPaymentHomeRemote", ProcessPaymentHomeRe-
                          mote.class);
            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
            return ticket;
        } catch(Exception e) {
            // EJB 1.0: throw new RemoteException("",e);
            throw new EJBException(e);
        }
    }
```

Этот метод иллюстрирует понятие рабочего потока. Он использует несколько компонентов, включая компоненты Reservation, ProcessPayment, Customer, Cabin и Cruise, для выполнения одной задачи: регистрации пассажира на круиз. Обманчиво простой, этот метод включает в себя несколько взаимодействий, которые вообще могли бы быть выполнены на стороне клиента. В ответ на один вызов клиентом метода `bookPassage()` компонент TravelAgent исполняет множество действий:

1. Ищет и получает ссылку на внутренний объект Reservation.
2. Создает новый компонент Reservation.
3. Ищет и получает удаленную ссылку на внутренний объект ProcessPayment.
4. Создает новый компонент ProcessPayment.
5. Принимает платеж с кредитной карточки пассажира, используя компонент ProcessPayment.
6. Генерирует новый `TicketDO` со всей необходимой информацией, описывающей покупку клиента.

С точки зрения проектирования заключение рабочего потока в сеансовый компонент с состоянием означает менее сложный интерфейс для клиента и большую гибкость при внесении изменений. Мы могли бы, например, легко изменить метод `bookPassage()` с целью проверки перекрывающихся заказов (когда клиент заказывает билеты на два разных круиза, которые накладываются по времени). Это расширение не приведет к изменению удаленного интерфейса и, следовательно, не потребует изменения клиентского приложения. Включение рабочего потока внутрь сеансовых компонентов с состоянием позволяет системе постоянно развиваться, никак не влияя на клиентов.

Кроме этого, может быть изменен тип поддерживаемых клиентов. Одна из самых больших проблем двухзвенных (two-tier) архитектур, кроме масштабируемости и управления транзакциями, – в том, что прикладная логика в них тесно переплетена с клиентской. Это усложняет многократное использование прикладной логики на разных клиентах. В случае применения сеансовых компонентов с состоянием такой проблемы нет, т. к. эти компоненты являются расширениями клиента, но не привязаны к его представлению. Предположим, что наша первая реализация системы резервирования использует Java-апплет с графическими элементами. Компонент TravelAgent мог бы управлять состоянием диалога и выполнять всю прикладную логику, тогда как апплет специализировался бы на графическом представлении. Если позднее мы решим перейти к «тонкому» клиенту (HTML, сгенерированный Java-сервлетом, например), мы просто могли бы использовать в сервлете этот же компонент TravelAgent. Поскольку вся прикладная логика заключена в сеансовом компоненте с состоянием, представление (Java-апплет, сервлет или что-то другое) может быть легко изменено.

Компонент TravelAgent также предоставляет транзакционную целостность для обработки резервирования пассажира. Если какая-либо операция внутри тела метода `bookPassage()` завершится неудачей, произойдет откат (roll back) всех этих операций, так что ни одно изменение не будет принято. Если компонент ProcessPayment не сможет принять платеж с кредитной карточки, то вновь создаваемый компонент Reservation и связанная с ним запись не будут созданы. Транзакционные аспекты компонента TravelAgent объясняются подробно в главе 14.

В EJB 2.0 удаленные и локальные компонентные ссылки могут использоваться внутри одного рабочего потока. Например, в методе `bookPassage()` доступ к компонентам Cruise и Cabin организован посредством локальных ссылок, а доступ к компонентам ProcessPayment и Customer – посредством удаленных. Такой механизм полностью приемлем. Контейнер EJB гарантирует атомарность транзакции, т. е. что неудача либо в удаленной, либо в локальной компонентной ссылке повлияет на всю транзакцию.

Для чего используется компонент Reservation?

Если у нас есть компонент Reservation, зачем нам нужен компонент TravelAgent? Хороший вопрос! Компонент TravelAgent использует компонент Reservation для создания заказа, но он также должен принимать платежи и генерировать билеты. Эти действия не имеют отношения к компоненту Reservation, поэтому они должны выполняться в сеансовом компоненте с состоянием, который может управлять рабочим потоком и областью транзакции. Кроме того, компонент TravelAgent реализует список, охватывающий несколько понятий системы «Титан». Было бы нелогично включать все эти возможности в компонент Reservation. (Для пользователей EJB 2.0: компонент Reservation был разработан в главе 7. Для тех, кто работает с EJB 1.1: код для этого компонента доступен на *http://www.oreilly.com/catalog/entjbeans3/*.)

ListAvailableCabins(): Реализация списка

Как и было обещано, мы собираемся вернуться к реализации списка кают, которую мы обыграли в главе 4. Однако на этот раз для получения списка мы не будем использовать компонент Cabin. Вместо этого мы обратимся непосредственно к базе данных. Обращение непосредственно к базе данных – это палка о двух концах. С одной стороны, нам не нужно обращаться непосредственно к базе данных, если существуют объектные компоненты, имеющие доступ к этой информации. Объектные компоненты предоставляют безопасный и логичный интерфейс для отдельного набора данных. После того как объектный компонент протестирован, он может использоваться во всех частях системы, значительно уменьшая вероятность возникновения проблем с целостностью данных. Примером такого механизма является компо-

нент Reservation. Объектные компоненты также могут собирать вместе разрозненные данные и использовать дополнительную прикладную логику, такую как проверка правильности, ограничений и безопасности для того, чтобы гарантировать, что доступ к данным соответствует прикладным правилам.

Но объектные компоненты не в состоянии описать каждый метод доступа к данным, который может потребоваться, они и не должны это делать. Одна из самых больших проблем, связанная с объектными компонентами, состоит в том, что они со временем имеют тенденцию раздуваться. Огромные компоненты, содержащие множества методов, – верный признак плохого дизайна. Объектные компоненты должны специализироваться на обеспечении доступа к очень ограниченному, но концептуально связанному набору данных. У вас должна быть возможность изменять, считывать и вставлять данные. Доступ к данным, охватывающий несколько понятий, таких как реализация списка, не следует включать в один объектный компонент.

Любой системе требуется реализация списка, для того чтобы дать своим клиентам возможность выбора. В системе резервирования, например, пассажиры должны выбрать каюту из списка свободных кают. Слово «*свободных*» является ключевым при определении этого поведения. Компонент Cabin может предоставить нам список кают, но он не знает, является ли каждая отдельная каюта свободной. Как вы, наверное, помните, отношение Cabin–Reservation, которое мы определили для EJB 2.0 в главе 7, было *однонаправленным*: компонент Reservation знал о своем отношении с Cabin, но не наоборот.

Вопрос о том, является ли каюта свободной, относится к использующему ее процессу – в нашем случае к компоненту TravelAgent – но не может относиться к самой каюте. В качестве аналогии: объект «Автомобиль» не заботится о том, на какой дороге он находится, он имеет дело только с характеристиками, описывающими его состояние и поведение. С другой стороны, система слежения за автотранспортом имеет дело с местоположением отдельных автомобилей.

Чтобы получить информацию о свободных каютах, мы должны сравнить список кают на нашем судне со списком кают, которые уже были забронированы. Метод `listAvailableCabins()` как раз это и делает. Он использует сложный запрос SQL для создания списка кают, которые еще не были зарезервированы для круиза, выбранного клиентом:

```
public String [] listAvailableCabins(int bedCount)
    throws IncompleteConversationalState {
    if (cruise == null)
        throw new IncompleteConversationalState();

    Connection con = null;
    PreparedStatement ps = null;;
    ResultSet result = null;
```

```
        try {
            Integer cruiseID = (Integer)cruise.getPrimaryKey();
            Integer shipID = (Integer)cruise.getShip().getPrimaryKey();
            con = getConnection();
            ps = con.prepareStatement(
                "select ID, NAME, DECK_LEVEL  from CABIN "+
                "where SHIP_ID = ? and BED_COUNT = ? and ID NOT IN "+
                "(SELECT CABIN_ID FROM RESERVATION "+" WHERE CRUISE_ID = ?)");

            ps.setInt(1,shipID.intValue());
            ps.setInt(2, bedCount);
            ps.setInt(3,cruiseID.intValue());
            result = ps.executeQuery();
            Vector vect = new Vector();
            while(result.next()) {
                StringBuffer buf = new StringBuffer();
                buf.append(result.getString(1));
                buf.append(',');
                buf.append(result.getString(2));
                buf.append(',');
                buf.append(result.getString(3));
                vect.addElement(buf.toString());
            }
            String [] returnArray = new String[vect.size()];
            vect.copyInto(returnArray);
            return returnArray;
        } catch (Exception e) {
            throw new EJBException(e);
        }
        finally {
            try {
                if (result != null) result.close();
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se){se.printStackTrace();}
        }
    }
```

Те, кто работает с EJB 1.1, используют для `listAvailableCabins()` **почти такой же код, но получают идентификатор компонента Ship иначе. Они должны заменить строку:**

```
Integer shipID = (Integer)cruise.getShip().getPrimaryKey();
```

на строку:

```
Integer shipID = cruise.getShipID ();
```

Это замена необходима из-за того, что EJB 1.1 не поддерживает поля отношений.

Как видите, запрос SQL достаточно сложен. Он мог бы быть определен с помощью метода, такого как `Cabin.findAvailableCabins(Cruise cruise)`**,**

в компоненте Cabin. Однако этот метод было бы сложно реализовать, поскольку компоненту Cabin потребуется обращаться к данным компонента Reservation. Другая причина для прямого доступа к базе данных в этом примере состоит в том, чтобы показать, что такой вид поведения нормален и, в некоторых случаях, предпочтителен. Иногда запрос является достаточно специфичным для конкретного сценария и не может многократно использоваться. Чтобы избежать добавления поисковых методов для каждого возможного запроса, можно просто использовать прямой доступ к базе данных, как показано в методе `listAvailableCabins()`. Прямой доступ к базе данных обычно меньше влияет на производительность, т. к. контейнеру не требуется иметь дело со ссылками на компонентные объекты, но он и менее пригоден для повторного использования. Все эти вопросы необходимо учитывать, принимая решение о том, следует ли запрос информации выполнять с помощью прямого доступа к базе данных или лучше определить новый поисковый метод.

Метод `listAvailableCabins()` возвращает удаленному клиенту массив объектов `String`. Мы могли бы возвращать коллекцию удаленных ссылок на Cabin, но мы этого не сделали. Причина проста: мы хотим сохранить клиентское приложение настолько легким, насколько это возможно. Список объектов `String` намного легче, чем его альтернатива – коллекция удаленных ссылок. Кроме того, применение коллекции удаленных ссылок потребовало бы от клиента работы с многими заглушками, причем каждая имела бы собственное соединение с компонентным объектом на сервере. Возвращая легкий массив `String`, мы уменьшаем число заглушек на стороне клиента, что сохраняет простоту клиента и экономит ресурсы сервера.

Для того чтобы этот метод заработал, необходимо создать метод `getConnection()` для получения соединения с базой данных и добавлять его к `TravelAgentBean`:

```
private Connection getConnection() throws SQLException {
    try {
        DataSource ds = (DataSource)jndiContext.lookup(
            "java:comp/env/jdbc/titanDB");
        return ds.getConnection();
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

Измените удаленный интерфейс компонента TravelAgent с целью включения метода `listAvailableCabins()`:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
```

```
import com.titan.processpayment.CreditCard;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(Integer cruise) throws RemoteException,
FinderException;

    public void setCabinID(Integer cabin) throws RemoteException,
FinderException;

    public TicketDO bookPassage(CreditCardDO card, double price)
        throws RemoteException,IncompleteConversationalState;

    public String [] listAvailableCabins(int bedCount)
        throws RemoteException, IncompleteConversationalState;
}
```

EJB 2.0: Дескриптор развертывания для TravelAgent

Следующий листинг представляет собой сокращенную версию XML-дескриптора развертывания, используемого для компонента TravelAgent. В нем определен не только компонент TravelAgent, но также и компоненты Customer, Cruise, Cabin и Reservation. Компонент ProcessPayment не определен в этом дескрипторе развертывания, потому что предполагается, что он будет развернут в отдельном файле JAR или, возможно, даже на сервере EJB, находящемся на другом сетевом узле:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <home>com.titan.travelagent.TravelAgentHomeRemote</home>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>

            <ejb-ref>
                <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
               <home>com.titan.processpayment.ProcessPaymentHomeRemote</
                    home>
                <remote>com.titan.processpayment.ProcessPaymentRemote</
                    remote>
            </ejb-ref>
            <ejb-local-ref>
                <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <local-home>com.titan.cabin.CabinHomeLocal</local-home>
                <local>com.titan.cabin.CabinLocal</local>
```

```
            </ejb-local-ref>
            <ejb-local-ref>
                <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
                <local>com.titan.cruise.CruiseLocal</local>
            </ejb-local-ref>
            <ejb-local-ref>
                <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <local-home>com.titan.reservation.ReservationHomeLocal</
                   local-home>
               <local>com.titan.reservation.ReservationLocal</local>
            </ejb-local-ref>

            <resource-ref>
                <description>DataSource for the Titan database</description>
                <res-ref-name>jdbc/titanDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
           </resource-ref>
        </session>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
            ...
        </entity>
        <entity>
            <ejb-name>CruiseEJB</ejb-name>
            <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
            <local>com.titan.cruise.CruiseLocal</local>
            ...
        </entity>
        <entity>
            <ejb-name>ReservationEJB</ejb-name>
           <local-home>com.titan.reservation.ReservationHomeLocal</local-home>
            <local>com.titan.reservation.ReservationLocal</local>
            ...
        </entity>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>Эта роль представляет всех</description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
```

```
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: Дескриптор развертывания для TravelAgent

При развертывании компонента TravelAgent в EJB 1.1 используйте следующий XML-дескриптор развертывания. Наиболее важные различия между этим дескриптором и дескриптором развертывания для компонента ProcessPayment – тег `<session-type>`, указывающий на то, что это компонент с состоянием, и применение элементов `<ejb-ref>` для описания компонентов, ссылки на которые доступны через ENC:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
    <enterprise-beans>
        <session>
            <description>
                Acts as a travel agent for booking passage on a ship.
            </description>
            <ejb-name>TravelAgentEJB</ejb-name>
            <home>com.titan.travelagent.TravelAgentHomeRemote</home>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>

            <ejb-ref>
                <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <home>com.titan.processpayment.ProcessPaymentHome</home>
                <remote>com.titan.processpayment.ProcessPayment</remote>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>ejb/CabinHomeRemote</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.cabin.CabinHome</home>
                <remote>com.titan.cabin.Cabin</remote>
            </ejb-ref>
            <ejb-ref>
```

```
                <ejb-ref-name>ejb/CruiseHomeRemote</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.cruise.CruiseHome</home>
                <remote>com.titan.cruise.Cruise</remote>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.customer.CustomerHome</home>
                <remote>com.titan.customer.Customer</remote>
            </ejb-ref>
            <ejb-ref>
                <ejb-ref-name>ejb/ReservationHomeRemote</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.titan.reservation.ReservationHome</home>
                <remote>com.titan.reservation.Reservation</remote>
            </ejb-ref>

            <resource-ref>
                <description>DataSource for the Titan database</description>
                <res-ref-name>jdbc/titanDB</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </session>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>Эта роль представляет всех</description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Сгенерировав дескриптор развертывания, заархивируйте компонент TravelAgent и разверните его на вашем сервере EJB. Вам также потребуется развернуть компоненты Reservation, Cruise и Customer, загруженные ранее. Опираясь на прикладные методы удаленного интерфейса компонента TravelAgent и прошлый опыт с компонентами Cabin, Ship и ProcessPayment, вы должны суметь создать собственное клиентское приложение для проверки этого кода.

📖 Рабочее упражнение 12.2. Сеансовый компонент с состоянием

Жизненный цикл сеансового компонента с состоянием

Самое большое различие между сеансовым компонентом с состоянием и другими типами компонентов состоит в том, что сеансовые компоненты с состоянием не используют пул экземпляров. Сеансовые компоненты с состоянием выделены одному клиенту на время всей своей жизни, поэтому для них не существует ни механизма подкачки, ни пула экземпляров.[1] Для того чтобы сохранить ресурсы, вместо перемещения в пул экземпляры сеансового компонента с состоянием просто удаляются из памяти. Связь компонентного объекта с клиентом сохраняется, но экземпляр компонента теряет все связанные с ним ссылки и удаляется сборщиком мусора в течение периода неактивности. Это означает, что каждый компонент с состоянием перед удалением должен быть пассивирован, для того чтобы сохранить состояние диалога экземпляра, а для восстановления состояния он должен быть активирован, когда компонентный объект снова становится активным.

Как компонент будет воспринимать свой жизненный цикл, зависит от того, реализует ли он специальный интерфейс `javax.ejb.SessionSynchronization`. Этот интерфейс определяет дополнительный набор методов обратного вызова, которые уведомляют компонент о его участии в транзакциях. Компонент, реализующий `SessionSynchronization`, перед внесением изменений может кэшировать данные из базы данных на протяжении нескольких вызовов методов. Мы еще не рассматривали транзакции подробно, поэтому не будем обсуждать эту часть жизненного цикла компонента до главы 14. Данный раздел описывает жизненный цикл сеансовых компонентов с состоянием, которые не реализуют интерфейс `SessionSynchronization`.

Жизненный цикл сеансового компонента с состоянием включает три состояния: «не существует», «готовых методов» и «пассивный». Это похоже на сеансовый компонент без состояния, но состояние готовых

[1] Некоторые производители используют пул сеансовых компонентов с состоянием, но это нестандартная реализация, которая не должна влиять на определенный жизненный цикл сеансового компонента с состоянием.

методов значительно отличается от пула готовых методов компонентов без состояния. На рис. 12.2 показана диаграмма состояний для сеансовых компонентов с состоянием.

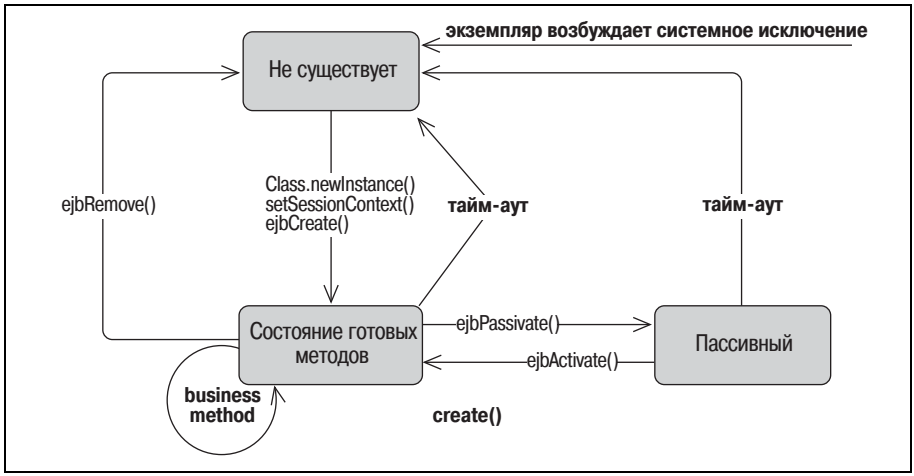

Рис. 12.2. Жизненный цикл сеансового компонента с состоянием

Состояние «не существует»

Экземпляр компонента с состоянием, находящийся в состоянии «не существует», не является экземпляром в памяти системы. Другими словами, он еще не был создан.

Состояние готовых методов

Состояние готовых методов – это состояние, в котором экземпляр компонента может обслуживать запросы от своих клиентов. В этом разделе рассматривается переход экземпляра в состояние готовых методов и обратно.

Переход в состояние готовых методов

Жизненный цикл компонента начинается, когда клиент вызывает метод `create()` внутреннего объекта сеансового компонента с состоянием. А когда вызов метода `create()` принимается контейнером, то контейнер вызывает метод `newInstance()` класса компонента, создавая новый экземпляр. Затем контейнер вызывает `setSessionContext()` экземпляра, вручая ему его ссылку на `SessionContext`, которую он должен хранить на протяжении всей жизни. В этот момент экземпляр компонента связывается со своим компонентным объектом. И наконец, контейнер вызывает метод `ejbCreate()` экземпляра, соответствующий методу `create()`, вызванному клиентом. После завершения `ejbCreate()` контейнер возвращает клиенту ссылку на компонентный объект. Экземпляр

теперь находится в состоянии готовых методов и готов к обслуживанию прикладных методов, вызываемых клиентом через удаленную ссылку компонента.

Жизнь в состоянии готовых методов

Находясь в состоянии готовых методов, экземпляр компонента может принимать вызовы методов от клиента. Это может быть связано с управлением рабочим потоком других компонентов или обращениями непосредственно к базе данных. В это время компонент может поддерживать в своих переменных состояние диалога и открытые ресурсы.

Переход из состояния готовых методов

Экземпляры компонента покидают состояние готовых методов при переходе или в пассивное состояние, или в состояние «не существует». В зависимости от того, как клиент использует компонент, от загрузки контейнера и алгоритма пассивации, применяемого производителем, экземпляр компонента может пассивироваться (и активироваться) несколько раз в течение своей жизни, а может и ни разу. Если компонент удаляется, он переводится в состояние «не существует». Компонент может быть удален либо клиентским приложением, через вызов одного из методов `remove()` клиентского API, либо контейнером.

Контейнер также может переместить экземпляр компонента из состо

яния готовых методов в состояние «не существует» при наступлении «тайм-аута» для компонента. «Тайм-ауты» задаются во время развертывания способом, специфичным для конкретного производителя. Если «тайм-аут» наступает в состоянии готовых методов, контейнер может, но не обязан, вызвать метод `ejbRemove()`. Компонент с состоянием не может быть удален во время выполнении транзакции.

Пассивное состояние

Во время жизни сеансового компонента с состоянием могут наступать периоды неактивности, в течение которых экземпляр компонента не обслуживает вызовы от клиента. Для экономии ресурсов контейнер может пассивировать экземпляр компонента во время его неактивности, сохраняя его состояние диалога и удаляя экземпляр компонента из памяти.

Когда компонент с состоянием пассивируется, поля экземпляра считываются, а затем записываются во внешнюю память, связанную с компонентным объектом. Если сеансовый компонент с состоянием успешно пассивирован, экземпляр удаляется из памяти: он разрушен.

Перед тем как компонент будет пассивироваться, вызывается его метод `ejbPassivate()`, уведомляя экземпляр компонента, что его собираются переводить в пассивное состояние. За это время экземпляр ком-

понента должен закрыть все открытые ресурсы и установить все неизменяемые несериализуемые поля в `null`. Это предотвращает появление проблем при сериализации компонента. Изменяемые (transient) поля просто будут игнорироваться.

Состояние диалога компонента может состоять только из примитивных значений, сериализуемых объектов и следующих специальных типов:

- EJB 2.0 и 1.1

  `javax.ejb.SessionContext`

  `javax.ejb.EJBHome` (типы удаленного внутреннего интерфейса)

  `javax.ejb.EJBObject` (типы удаленного интерфейса)

  `javax.jta.UserTransaction` (интерфейс транзакции компонента)

  `javax.naming.Context` (только если он ссылается на JNDI ENC)

- только EJB 2.0

  `javax.ejb.EJBLocalHome` (типы локального внутреннего интерфейса)

  `javax.ejb.EJBLocalObject` (типы локального интерфейса)

  Ссылки на управляемые фабрики ресурсов (например, `javax.sql.DataSource`)

Типы из этого списка (и их подтипы) обрабатываются механизмом пассивации особо. Они не должны быть сериализуемыми, они будут автоматически обработаны при пассивации и восстановлены в экземпляре компонента, когда он будет активироваться.

Состояние диалога экземпляра компонента записывается во внешнюю память, чтобы сохранить его, когда экземпляр будет пассивирован и разрушен. Для сохранения экземпляра компонента контейнеры могут использовать стандартную сериализацию Java или некоторый другой механизм, приводящий к такому же результату. Некоторые производители, например, просто считывают значения полей и сохраняют их в кэше. От контейнера требуется сохранить вместе с состоянием диалога удаленные ссылки на другие компоненты. Когда компонент активируется, контейнер должен автоматически восстановить все компонентные ссылки. Контейнер также должен восстановить все ссылки на специальные типы, перечисленные выше.

Поля, объявленные как `transient`, не будут сохраняться при пассивации компонента. За исключением специальных типов, перечисленных выше, все неизменяемые и несериализуемые поля перед пассивацией экземпляра должны быть установлены в `null`, иначе контейнер уничтожит экземпляр компонента, сделав его недоступным для последующего использования клиентом. Ссылки на специальные типы должны автоматически сохраняться контейнером вместе с сериализуемым экземпляром компонента так, чтобы они могли быть восстановлены при активации компонента.

Когда клиент выполняет запрос к компонентному объекту, чей компонент пассивен, контейнер активирует данный экземпляр. Этот процесс включает в себя десериализацию экземпляра компонента и восстановление ссылки на `SessionContext`, компонентных ссылок и управляемых фабрик ресурсов (только в EJB 2.0), хранившихся в экземпляре до его пассивации. После успешного восстановления состояния диалога компонента вызывается метод `ejbActivate()`. Внутри данного метода экземпляр компонента должен открыть все ресурсы, которые не могут быть пассивированы, и инициализировать значения всех изменяемых полей. По завершении `ejbActivate()` компонент возвращается назад в состояние готовых методов и становится доступным для обслуживания клиентских запросов, направляемых через компонентный объект.

В EJB 1.1 открытые ресурсы, такие как сетевые соединения и соединения с JDBC, должны закрываться каждый раз, когда компонент пассивируется. В сеансовых компонентах с состоянием открытые ресурсы не будут поддерживаться в течение всей жизни экземпляра. Когда сеансовый компонент с состоянием пассивируется, любой открытый ресурс может вызывать проблемы в механизме активации.

Активация экземпляра компонента подчиняется правилам сериализации Java. Исключение представляют собой изменяемые поля. В сериализации Java при десериализации объекта изменяемые поля устанавливаются в свои значения по умолчанию. Значения полей примитивных типов станут нулевыми, поля `Boolean` примут значение `false`, а объектные ссылки – `null`. В EJB изменяемые поля не обязаны быть установленными в их начальные значения, поэтому при активации компонента они могут содержать произвольные значения. Значения, содержащиеся в изменяемых полях после активации, непредсказуемы в разных реализациях, поэтому не стоит рассчитывать, что они будут инициализированы. Лучше вызовите метод `ejbActivate()` для присвоения им значений.

Контейнер также может переместить экземпляр компонента из пассивного состояния в состояние «не существует», если для компонента наступит «тайм-аут». Если «тайм-аут» происходит в пассивном состоянии, метод `ejbRemove()` не вызывается.

Системные исключения

Каждый раз, когда в методе компонента возбуждается системное исключение, контейнер делает недействительным компонентный объект и уничтожает экземпляр компонента. Экземпляр компонента пе-

ремещается непосредственно в состояние «не существует», а метод `ejbRemove()` не вызывается.

Системное исключение – это любое неперехватываемое исключение, в том числе и `EJBException`. Перехватываемые исключения, возбуждаемые в подсистемах, обычно оборачиваются в `EJBException` и повторно возбуждаются в виде системных исключений. Перехватываемое исключение, возбужденное подсистемой, не следует обрабатывать таким способом, если компонент может безопасно после него восстановиться. Однако в большинстве случаев исключение подсистемы должно быть повторно возбуждено в виде `EJBException`.

В EJB 1.1 исключение `java.rmi.RemoteException` для обратной совместимости с EJB 1.0 также рассматривается как системное исключение. Однако возбуждение `RemoteException` в методе класса компонента было отменено и объявлено нерекомендуемым.

13

Компоненты, управляемые сообщениями

Эта глава разбита на два подраздела: «JMS как средство» и «Компоненты, управляемые сообщениями». Первый раздел описывает службу сообщений Java (Java Message Service, JMS) и ее роль в качестве ресурса, доступного всем типам компонентов (сеансовым, объектным и управляемым сообщениями). Читателям, не знакомым с JMS, следует прочитать первый раздел и только потом переходить ко второму.

Те же, для кого JMS не является тайной за семью печатями, могут перейти прямо ко второму разделу, содержащему краткий обзор нового типа компонента – компонента, управляемого сообщениями. Компонент, управляемый сообщениями, – это асинхронный компонент, активируемый при получении сообщения. В EJB 2.0 производители обязаны поддерживать компоненты, управляемые сообщениями на базе JMS, принимающие JMS-сообщения из отдельных тем (topics) или очередей (queues) и обрабатывающие эти сообщения по мере их поступления.

Все производители EJB 2.0 должны по умолчанию поддерживать провайдер JMS. Большая часть производителей EJB 2.0 имеют встроенный провайдер JMS, а некоторые кроме него могут поддерживать и других провайдеров JMS. Независимо от того, каким образом производитель EJB 2.0 предоставляет службу JMS, она является обязательной, если производитель предполагает поддерживать компоненты, управляемые сообщениями. Преимущество этого принудительного включения JMS состоит в том, что разработчики EJB могут рассчитывать на присутствие работоспособного провайдера JMS, с помощью которого можно как отправлять, так и передавать сообщения.

JMS как средство

JMS – это стандартный, независимый от производителя программный интерфейс (API), который является частью платформы J2EE и может применяться для доступа к корпоративным системам передачи сообщений. Корпоративные системы передачи сообщений (ориентируемые на сообщения программные продукты) дают возможность обмениваться сообщениями между приложениями через сеть. JMS во многом схож с JDBC: так же, как JDBC является программным интерфейсом, который может служить для доступа ко многим различным реляционным базам данных, JMS предоставляет такой же независимый от производителей доступ к корпоративным системам передачи сообщений. Многие программы передачи сообщений на уровне предприятия в настоящее время поддерживают JMS, включая такие, как MQSeries от IBM, службу JMS WebLogic от BEA, iPlanet Message Queue от Sun Microsystems, SonicMQ от Progress и другие. Приложения, в которых посылка или прием сообщений основываются на JMS API, переносимы между разными марками JMS-продуктов.

Прикладные программы Java, использующие JMS, называются *клиентами JMS* (*JMS clients*), а система передачи сообщений, управляющая маршрутизацией и доставкой сообщений, называется *провайдером JMS* (*JMS provider*). *Приложение JMS* (*JMS application*) – это прикладная система, состоящая из нескольких JMS-клиентов и обычно одного JMS-провайдера.

JMS-клиент, посылающий сообщение, называется *поставщиком* (*producer*), а JMS-клиент, принимающий сообщение, называется *потребителем* (*consumer*). Один JMS-клиент может быть одновременно и поставщиком и потребителем. Когда мы употребляем термины «потребитель» и «поставщик», мы подразумеваем JMS-клиент, который соответственно принимает или передает сообщения.

В EJB компоненты всех типов могут использовать JMS для передачи сообщений различным адресатам. Эти сообщения принимаются другими прикладными программами Java или компонентами, управляемыми сообщениями. JMS обеспечивает передачу сообщений от компонентов с помощью службы передачи сообщений, иногда называемой посредником сообщения или маршрутизатором. Посредники сообщений существуют уже пару десятилетий (самым старым и наиболее устоявшимся является MQSeries от IBM), но сама JMS достаточно нова и специально предназначена для передачи различных типов сообщений от одного приложения Java другому.

Новая реализация компонента TravelAgent с использованием JMS

Мы можем изменить компонент TravelAgent, разработанный в главе 12, так, чтобы он использовал JMS для уведомления некоторого другого

приложения Java о том, что был сделан заказ билетов. Приведенный ниже код показывает, как следует изменить метод bookPassage(), чтобы компонент TravelAgent посылал простое текстовое сообщение, основанное на описании объекта TicketDO:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price, new Date());

        Object ref = jndiContext.lookup
            ("java:comp/env/ejb/ProcessPaymentHomeRemote");

        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
           PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

        String ticketDescription = ticket.toString();

        TopicConnectionFactory factory = (TopicConnectionFactory)
            jndiContext.lookup("java:comp/env/jms/TopicFactory");

        Topic topic = (Topic)
            jndiContext.lookup("java:comp/env/jms/TicketTopic");

        TopicConnection connect = factory.createTopicConnection();

        TopicSession session = connect.createTopicSession(true,0);

        TopicPublisher publisher = session.createPublisher(topic);

        TextMessage textMsg = session.createTextMessage();
        textMsg.setText(ticketDescription);
        publisher.publish(textMsg);
        connect.close();

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

Посылка сообщения потребовала добавления довольно большого объема нового кода. Несмотря на этот поначалу пугающий факт, основы JMS не так уж и сложны.

TopicConnectionFactory и Topic

Для того чтобы посылать сообщение JMS, нам необходимо соединение с провайдером JMS и адрес назначения сообщения. Подключение к провайдеру JMS может быть выполнено через фабрику подключений JMS, а адрес места назначения сообщения определяется объектом `Topic` (тема). И фабрика подключений и объект `Topic` извлекаются из JNDI ENC компонента TravelAgent:

```
TopicConnectionFactory factory = (TopicConnectionFactory)
    jndiContext.lookup("java:comp/env/jms/TopicFactory");

Topic topic = (Topic)
    jndiContext.lookup("java:comp/env/jms/TicketTopic");
```

Фабрика `TopicConnectionFactory` в JMS по своим функциям похожа на `DataSource` в JDBC. Так же как `DataSource` предоставляет JDBC-подключение к базе данных, `TopicConnectionFactory` предоставляет JMS-подключение к маршрутизатору сообщений.[1]

Сам объект `Topic` представляет независимый от сетевой реализации пункт назначения, в который будет отправлено сообщение. В JMS сообщения посылаются в пункт назначения (тему или очередь), а не другим прикладным программам непосредственно. Тема представляет собой аналог списка рассылки или телеконференции: любое приложение, обладающее достаточными правами, может принимать сообщения из темы и посылать сообщения в тему. Когда клиент JMS принимает сообщения из темы, говорят, что клиент *подписан (subscribed)* на данную тему. JMS разделяет прикладные программы, позволяя им посылать сообщения друг другу через пункт назначения, который работает в качестве виртуального канала.

JMS также поддерживает другой тип пункта назначения, называемый *очередью (queue)*. Различия между темами и очередями более подробно объяснены ниже.

TopicConnection и TopicSession

`TopicConnectionFactory` предназначена для создания `TopicConnection`, который является текущим подключением к провайдеру JMS:

```
TopicConnection connect = factory.createTopicConnection();

TopicSession session = connect.createTopicSession(true,0);
```

После того как `TopicConnection` получен, он может использоваться для создания `TopicSession`. `TopicSession` позволяет Java-программисту сгруппировать операции посылки и приема сообщений. В нашем случае нам

[1] Аналогия не совсем точная. Можно было бы сказать, что `TopicSession` являются аналогом `DataSource`, т. к. они оба представляют соединения с ресурсами транзакций.

понадобится лишь один TopicSession. Однако часто полезно иметь несколько объектов TopicSession: если требуется создавать и получать сообщения с применением многопоточности, каждым потоком должен быть создан отдельный объект Session, относящийся к данному потоку. Это необходимо ввиду того, что объекты JMS Session основываются на однопоточной модели, запрещающей параллельный доступ нескольких потоков к одному объекту Session. Поток, создавший TopicSession, – это обычно поток, который использует производителей и потребителей данного объекта Session (т. е. объекты TopicPublisher и TopicSubscriber). Если необходимо создавать и получать сообщения на основе многопоточности, каждым потоком должен быть создан и использован отдельный объект Session.

Метод createTopicSession() имеет два параметра:

```
createTopicSession(boolean transacted, int acknowledgeMode)
```

В соответствии со спецификацией EJB 2.0 эти параметры игнорируются во время выполнения, т. к. контейнер EJB управляет режимами транзакций и подтверждений каждого ресурса JMS, полученного через JNDI ENC. Спецификация рекомендует, чтобы разработчики использовали параметры true для transacted и 0 для acknowlegeMode, но поскольку они, по определению, игнорируются, не имеет значения, что вы будете использовать. К сожалению, не все производители твердо придерживаются этой части спецификации. Одни производители игнорируют данные параметры, а другие – нет. Ознакомьтесь с документацией от производителя, чтобы определить правильные значения для этих параметров и в управляемых контейнером, и в управляемых компонентом транзакциях.

Хорошим стилем программирования является закрытие TopicConnection по окончании работы с ним, т. к. это позволяет сберечь ресурсы:

```
TopicConnection connect = factory.createTopicConnection();
...

connect.close();
```

TopicPublisher

TopicSession применяется для создания TopicPublisher, посылающего сообщения из компонента TravelAgent в пункт назначения, указанный объектом Topic. Каждый клиент JMS, подписавшийся на данную тему, получит копию сообщения:

```
TopicPublisher publisher = session.createPublisher(topic);

TextMessage textMsg = session.createTextMessage();
textMsg.setText(ticketDescription);
publisher.publish(textMsg);
```

Типы сообщений

Сообщение в JMS – это объект Java, состоящий из двух частей: заголовка и тела сообщения. В заголовке находится информация о доставке и метаданные, а тело сообщения заключает в себе данные прикладной программы, которые могут принимать разные формы: текстовые, сериализуемые объекты, байтовые потоки и т. д. JMS API определяет несколько типов сообщений (`TextMessage`, `MapMessage`, `ObjectMessage` и другие) и предоставляет методы для доставки и приема сообщений от других прикладных программ.

Например, мы можем модифицировать компонент TravelAgent так, чтобы он вместо `TextMessage` посылал `MapMessage`:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

MapMessage mapMsg = session.createTextMessage();
mapMsg.setInt("CustomerID", ticket.customerID.intValue());
mapMsg.setInt("CruiseID", ticket.cruiseID.intValue());
mapMsg.setInt("CabinID", ticket.cabinID.intValue());
mapMsg.setDouble("Price", ticket.price);

publisher.publish(mapMsg);
```

JMS-клиенты, которые примут `MapMessage`, могут обращаться к его атрибутам (`CustomerID`, `CruiseID`, `CabinID` и `Price`) по имени.

В качестве альтернативы компонент TravelAgent мог бы быть изменен так, чтобы использовать тип `ObjectMessage`, который позволит нам посылать в виде сообщения весь объект `TicketDO`, с применением сериализации Java:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

ObjectMessage objectMsg = session.createObjectMessage();
ObjectMsg.setObject(ticket);

publisher.publish(objectMsg);
```

Кроме `TextMessage`, `MapMessage` и `ObjectMessage` JMS предоставляет еще два типа сообщений: `StreamMessage` и `BytesMessage`. Первый из них – `StreamMessage` – может нести в качестве полезной нагрузки содержимое потока ввода-вывода. А `BytesMessage` может принимать любой массив байт, которые он воспринимает как необрабатываемые данные.

XML-дескриптор развертывания

Применяя JMS, это средство необходимо объявить в XML-дескрипторе развертывания компонента, таким же образом, как и ресурс JDBC, используемый компонентом Ship в главе 10:

```
<enterprise-beans>
    <session>
        <ejb-name>TravelAgentBean</ejb-name>
        ...
        <resource-ref>
            <res-ref-name>jms/TopicFactory</res-ref-name>
            <res-type>javax.jms.TopicConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
        <resource-ref>
            <res-ref-name>jdbc/titanDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
        <resource-env-ref>
            <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
            <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
        </resource-env-ref>
        ...
    </session>
```

Элемент `<resource-ref>` для JMS `TopicConnectionFactory` похож на объявление `<resource-ref>` для JDBC `DataSource`: он объявляет имя JNDI ENC, тип интерфейса и протокол авторизации. Кроме `<resource-ref>` компонент TravelAgent также должен объявить элемент `<resource-env-ref>`, в котором перечислены все «управляемые объекты», связанные с элементом `<resource-ref>`. В данном случае мы объявляем `Topic`, используемый для отправки сообщения о билете. Во время развертывания JMS `TopicConnectionFactory` и `Topic`, объявленные в элементах `<resource-ref>` и `<resource-env-ref>`, необходимо связать с JMS-фабрикой и темой.

Прикладной клиент JMS

Для того чтобы лучше понять механизм применения JMS, мы создадим приложение Java, единственным назначением которого будет прием и обработка сообщений о заказах билетов. Разработаем очень простой клиент JMS, просто выводящий описание по каждому билету по мере поступления сообщений. Допустим, что компонент TravelAgent для посылки описания билета клиентам JMS использует `TextMessage`. Следующий код показывает, как могло бы выглядеть клиентское приложение JMS:

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.Session;
import javax.jms.TopicSubscriber;
```

```
import javax.jms.JMSException;
import javax.naming.InitialContext;
public class JmsClient_1 implements javax.jms.MessageListener {

    public static void main(String [] args) throws Exception {

        if(args.length != 2)
            throw new Exception("Недопустимое количество аргументов");

        new JmsClient_1(args[0], args[1]);

        while(true){Thread.sleep(10000);}

    }

   public JmsClient_1(String factoryName, String topicName) throws Exception
{

        InitialContext jndiContext = getInitialContext();

        TopicConnectionFactory factory = (TopicConnectionFactory)
            jndiContext.lookup("ИмяФабрикиТемы");

        Topic topic = (Topic)jndiContext.lookup("ИмяТемы");

        TopicConnection connect = factory.createTopicConnection();

        TopicSession session =
            connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

        TopicSubscriber subscriber = session.createSubscriber(topic);

        subscriber.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message) {
        try {

            TextMessage textMsg = (TextMessage)message;
            String text = textMsg.getText();
            System.out.println("\n RESERVATION RECIEVED:\n"+text);

        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }

    public static InitialContext getInitialContext() {
        // Создаем контекст JNDI, специфичный для производителя
    }
}
```

Конструктор `JmsClient_1` **получает** `TopicConnectionFactory` **и** `Topic` **из** `InitialContext` **JNDI. Этот контекст создается со специфичными для производителя свойствами так, чтобы клиент мог соединиться с тем же**

провайдером JMS, который используется компонентом TravelAgent. Например, метод getInitialContext() для сервера приложений WebLogic будет выглядеть следующим образом:[1]

```
public static InitialContext getInitialContext() {
    Properties env = new Properties();
    env.put(Context.SECURITY_PRINCIPAL, "guest");
    env.put(Context.SECURITY_CREDENTIALS, "guest");
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
}
```

Клиент, получив TopicConnectionFactory и Topic, создает TopicConnection и TopicSession таким же образом, как и компонент TravelAgent. Главное отличие состоит в том, что объект TopicSession применяется для создания TopicSubscriber вместо TopicPublisher. TopicSubscriber специально предназначен для обработки входящих сообщений, созданных для указанной в нем темы:

```
TopicSession session =
    connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

TopicSubscriber subscriber = session.createSubscriber(topic);

subscriber.setMessageListener(this);

connect.start();
```

TopicSubscriber может принимать сообщения непосредственно или перенаправлять обработку сообщений объекту javax.jms.MessageListener. Мы решили, что JmsClient_1 будет реализовывать интерфейс MessageListener так, чтобы он мог обрабатывать сообщения самостоятельно. Объекты MessageListener реализуют единственный метод – onMessage(), вызываемый каждый раз, когда в тему подписчика посылается новое сообщение. В нашем случае каждый раз, когда компонент TravelAgent посылает сообщение о резервировании билетов в тему, у клиента JMS будет вызываться его метод onMessage() так, чтобы он мог принять копию сообщения и обработать его:

```
public void onMessage(Message message) {
    try {
        TextMessage textMsg = (TextMessage)message;
        String text = textMsg.getText();
        System.out.println("\n Получена информация о резервации:\n"+text);
```

[1] Кроме этого, JNDI позволяет устанавливать свойства в файле *jndi.properties*, содержащем значения свойств для InitialContext, и может быть обработан динамически во время выполнения. В этой книге свойство устанавливается явно.

```
        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }
```

📖 Рабочее упражнение 13.1. JMS как средство

JMS – асинхронная система

Одно из основных преимуществ системы сообщений JMS состоит в том, что она является *асинхронной* (*asynchronous*). Другими словами, клиент JMS может посылать сообщение без необходимости ждать ответ. Сравните эту гибкость с системой синхронных сообщений Java RMI. RMI – это превосходный выбор для сборки транзакционных компонентов, но для некоторых применений ее возможности слишком ограничены. Клиент, вызывая метод компонента, каждый раз блокирует текущий поток, пока метод не завершит выполнение. Эта обработка с блокировкой делает клиент зависимым от готовности сервера EJB и приводит к тесной связи между клиентом и компонентом.

В JMS клиент асинхронно посылает сообщения в пункт назначения (тему или очередь), из которого другие клиенты JMS одновременно могут принимать сообщения. Когда клиент JMS посылает сообщение, то не ждет ответа. Он посылает сообщение маршрутизатору, отвечающему за доставку его другим клиентам. Клиенты, посылающие сообщения, отделены от клиентов, принимающих их. Отправители не зависят от готовности получателей.

Ограничения RMI делают JMS привлекательной альтернативой для связи с другими приложениями. Используя стандартный контекст имен окружения JNDI, компонент может получать JMS-соединение с провайдером JMS и с его помощью организовывать доставку асинхронных сообщений другим приложениям Java. Например, сеансовый компонент TravelAgent может средствами JMS уведомлять другие приложения об окончании обработки заказа, как показано на рис. 13.1.

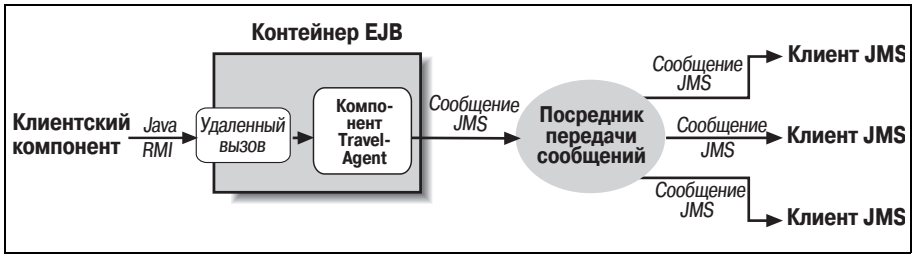

Рис. 13.1. Использование JMS в компоненте TravelAgent

В данном случае приложения, принимающие сообщения JMS от компонента TravelAgent, могут быть компонентами, управляемыми со-

общениями, другими корпоративными приложениями Java или приложениями других организаций, которым нужна информация о том, что заказ был обработан. Примеры могли бы включать и деловых партнеров, одновременно использующих информацию о клиенте, и внутреннее маркетинговое приложение, заносящее клиентов в список рассылки.

JMS дает возможность компоненту послать сообщения без блокировки. Компонент не знает, кто примет сообщение, из-за того, что он передает сообщение виртуальному каналу (пункту назначения), а не другому приложению непосредственно. Прикладные программы могут выбрать получение сообщения из этого виртуального канала и принимать уведомления о новых заказах билетов.

Один интересный аспект корпоративной системы сообщений состоит в том, что развязанный асинхронный характер этой технологии означает, что контексты транзакций и безопасности отправителя не наследуются получателем сообщения. Например, если компонент TravelAgent посылает сообщение о билете, оно может быть авторизовано провайдером JMS, но его контекст безопасности не будет передан клиенту JMS, принимающему сообщение. Когда клиент JMS будет принимать сообщение от компонента TravelAgent, он не будет иметь никакого представления о контексте безопасности, под которым данное сообщение было послано. Так и должно быть, потому что отправитель и получатель часто работают в разных окружениях с разными зонами безопасности.

Точно так же транзакции никогда не передаются от отправителя получателю. С одной стороны, отправитель понятия не имеет о том, кто получает его сообщения. Если сообщение посылается в тему, у него может быть один или тысячи получателей. Управление распределенной транзакцией в таких неопределенных ситуациях недостаточно надежно. Кроме того, клиенты, принимающие сообщение, могут не получить его в течение долгого времени после того, как оно было послано, в связи с тем, что они временно были отключены или по другой причине неспособны были принимать сообщения. Одно из основных преимуществ JMS состоит в том, что он допускает временное рассоединение отправителей и получателей. Транзакции предназначены для быстрого выполнения, поскольку они блокируют ресурсы. Длинная транзакция с непредсказуемым завершением также является ненадежной.

Клиент JMS, однако, может иметь распределенную транзакцию с провайдером JMS так, чтобы управлять операциями посылки и получения в контексте транзакции. Например, если транзакция компонента TravelAgent прервется по какой-либо причине, провайдер JMS исключит сообщение о билете, посланное компонентом TravelAgent. Транзакции и JMS более подробно рассматриваются в главе 14.

Модели передачи сообщений JMS: «издание–подписка» и «точка–точка»

JMS предоставляет два типа моделей передачи сообщений*: «издание-подписка»* (*publish-and-subscribe*) и «*точка-точка*» (*point-to-point*). Спецификация JMS называет их *зонами сообщений* (*messaging domains*). В терминологии JMS «издание-подписка» и «точка-точка» часто сокращаются до pub/sub и p2p (или PTP), соответственно. В данной главе используются и длинные и короткие формы записи.

В двух словах, «издание-подписка» предназначена для передачи сообщений типа «один-ко-многим», тогда как модель «точка-точка» предназначена для рассылки сообщений «один-к-одному» (рис. 13.2).

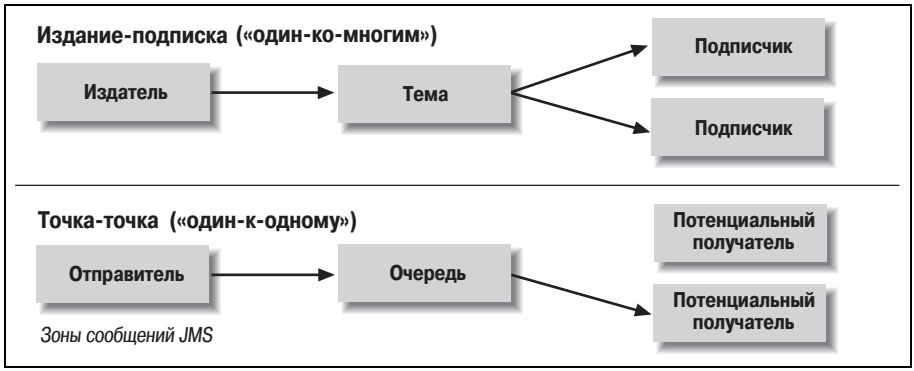

Рис. 13.2. Зоны сообщений JMS

«Издание–подписка»

При передаче сообщений «издание-подписка» один поставщик может посылать сообщение многим потребителям через виртуальный канал, называемый *темой* (*topic*). Потребители могут выбрать *подписку* (*subscribe*) на любую тему. Все сообщения, направляемые в тему, передаются всем потребителям данной темы. Каждый потребитель принимает копию каждого сообщения. Модель передачи сообщений pub/sub, по существу, представляет собой модель, основанную на проталкивании (push-based), где сообщения автоматически распространяются по потребителям без необходимости запрашивать, или вытаскивать, новые сообщения из темы.

В модели передачи сообщений pub/sub производитель, посылающий сообщение, не зависит от потребителей, принимающих сообщение. Дополнительно клиенты JMS, использующие pub/sub, могут устанавливать долговременные подписки, позволяющие потребителям отсоединиться и позже снова подключиться и забрать сообщения, поступившие во время отключения связи.

Компонент TravelAgent в данной главе основывается на программной модели pub/sub с объектом `Topic` в качестве пункта назначения.

«Точка-точка»

Модель передачи сообщений «точка-точка» позволяет клиентам JMS посылать и принимать сообщения как синхронно, так и асинхронно, через виртуальные каналы, известные как *очереди* (*queues*). Модель передачи сообщений p2p традиционно основывается на модели вытягивания (pull), или опроса (poll), в которой сообщения запрашиваются из очереди вместо автоматического выталкивания клиенту.[1]

У очереди может быть несколько получателей, но только один получатель может принимать каждое отдельное сообщение. Как было показано выше на рис. 13.2, провайдер JMS заботится о раздаче сообщений клиентам JMS, гарантируя, что каждое сообщение будет использовано только одним клиентом. Спецификация JMS не задает правила распределения сообщений среди нескольких получателей.

Программный интерфейс службы сообщений для p2p похож на используемый для pub/sub. Следующий фрагмент кода содержит листинг, иллюстрирующий, как компонент TravelAgent мог бы быть изменен для использования основанного на очереди API для p2p вместо основанной на теме модели pub/sub, фигурирующей в приведенном ранее примере:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
        ...

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);


        String ticketDescription = ticket.toString();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("java:comp/env/jms/QueueFactory");

        Queue queue = (Queue)
            jndiContext.lookup("java:comp/env/jms/TicketQueue");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session = connect.createQueueSession(true,0);

        QueueSender sender = session.createSender(queue);

        TextMessage textMsg = session.createTextMessage();
        textMsg.setText(ticketDescription);
        sender.send(textMsg);
        connect.close();
```

[1] В спецификации JMS не указывается определенно, как должны быть реализованы модели p2p и pub/sub. Каждая из них может использовать и проталкивание и опрос, но, по крайней мере концептуально, модель pub/sub основана на проталкивании, а модель p2p – на опросе.

```
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
```

Какую модель сообщений предпочесть?

Причины существования двух моделей объясняются происхождением спецификации JMS. JMS начиналась как способ обеспечения общего API для доступа к существующим системам передачи сообщений. Во время его планирования одни производители придерживались модели p2p передачи сообщений, а другие – модели pub/sub. Следовательно, для получения широкой промышленной поддержки в JMS требовалось предоставить API для обеих моделей. Спецификация JMS 1.0.2 не требовала, чтобы провайдер JMS поддерживал обе модели. Однако от производителя EJB 2.0 требуется поддержка обеих моделей передачи сообщений.

Почти все, что может быть сделано при помощи модели pub/sub, можно сделать и с помощью «точка-точка», и наоборот. Аналогия может быть проведена с предпочтением разработчиками различных языков программирования. Теоретически любое приложение, которое может быть написано на Паскале, может также быть написано и на C. Все что может быть написано на C++, может также быть написано и на Java. В некоторых случаях все сводится к вопросу о предпочтениях или к тому, с какой моделью вы уже знакомы.

В большинстве случаев решение о том, какую модель предпочесть, зависит от различных достоинств каждой модели. В случае с pub/sub тему может прослушивать любое число подписчиков, и все они будут принимать копии одних и тех же сообщений. Поставщик не заботится о том, слушает ли кто-нибудь его. Например, рассмотрим издателя, который рассылает биржевые котировки. Если какой-либо отдельный подписчик в данное время не подсоединен и пропустил выгодную цену, издателя это не касается. И наоборот, сессия «точка-точка», вероятно, будет предназначена для диалога «один-к-одному» с отдельным приложением, находящемся на другом конце. В этом сценарии каждое сообщение действительно важно.

Диапазон и разнообразие данных, представляемых сообщениями, также могут сыграть свою роль. В случае применения pub/sub сообщения распределяются по потребителям, основываясь на фильтрации, которая задается использованием определенных тем. Даже если сообщения применяются для установки связи «один-к-одному» с другим известным приложением, может быть выгодно использовать pub/sub с несколькими темами для выделения разных видов сообщений. Каждый вид сообщения может обрабатываться отдельно своим собственным уникальным потребителем и слушателем `onMessage()`.

Модель «точка-точка» более удобна, если требуется, чтобы отдельный получатель обработал данное сообщение только один раз. Возможно, это наиболее существенное различие между двумя моделями: p2p гарантирует, что только один потребитель обработает каждое сообщение. Это чрезвычайно важно, когда сообщения должны быть обработаны отдельно и последовательно друг за другом.

Объектные и сеансовые компоненты не должны принимать сообщения

JmsClient_1 был предназначен для получения сообщений, производимых компонентом TravelAgent. Может ли другой объектный или сеансовый компонент также принимать эти сообщения? Ответ – да, но это – очень плохая мысль.

Объектные и сеансовые компоненты отвечают на вызовы Java RMI от компонентных клиентов и не могут быть запрограммированы, чтобы отвечать на сообщения JMS так же, как это делают компоненты, управляемые сообщениями. Это означает, что невозможно создать сеансовый или объектный компонент, который будет управляться входящими сообщениями. Неспособность делать компонент, отвечающий на сообщения JMS, послужило причиной того, что в EJB 2.0 были введены компоненты, управляемые сообщениями. Компоненты, управляемые сообщениями, предназначены для получения сообщений из тем и очередей. Они заполнили важную нишу, и в следующем разделе мы подробно изучим, как их программировать.

Можно разработать объектный или сеансовый компонент, который может получать сообщение JMS из прикладного метода, но сначала этот метод должен быть вызван клиентом EJB. Например, вызываемый прикладной метод компонента Hypothetical устанавливает сессию JMS и затем пытается считать сообщение из очереди:

```
public class HypotheticalBean implements javax.ejb.SessionBean {
    InitialContext jndiContext;

    public String businessMethod() {

        try{

            QueueConnectionFactory factory = (QueueConnectionFactory)
                jndiContext.lookup("java:comp/env/jms/QueueFactory");

            Queue topic = (Queue)
                jndiContext.lookup("java:comp/env/jms/Queue");

            QueueConnection connect = factory.createQueueConneciton();
            QueueSession session = connect.createQueueSession(true,0);
            QueueReceiver receiver = session.createReceiver(queue);

            TextMessage textMsg = (TextMessage)receiver.receive();
```

```
        connect.close();

                return textMsg.getText();

            } catch(Exception e) {
                throws new EJBException(e);
            }

        }
        ...
    }
```

Объект `QueueReceiver`, являющийся потребителем сообщений, использует упреждающее чтение сообщения из очереди. Хотя все было запрограммировано правильно, это – достаточно опасная операция, т. к. вызов метода `QueueReceiver.receive()` блокирует поток, пока сообщение не станет доступным. Если сообщение никогда не поступит в очередь получателя, поток будет блокирован на неопределенное время! Другими словами, если никто никогда не пошлет сообщение в очередь, `QueueReceiver` будет просто сидеть там в вечном ожидании.

По правде говоря, существуют также и другие методы `receive()`, которые являются менее опасными. Например, `receive(long timeout)` позволяет указать время, после которого `QueueReceiver` должен снять блокировку потока и прекратить ожидание сообщения. Также существует метод `receiveNoWait()`, проверяющий наличие сообщений и возвращающий `null`, если нет ни одного ожидающего сообщения, таким образом избегая длительной блокировки потока.

Хотя альтернативные методы `receive()` гораздо безопаснее, применение их все же рискованно. Нет никакой гарантии, что они будут выполняться, как мы ожидаем, и риск ошибки программиста (например, использование неверного метода `receive()`) слишком велик. Кроме этого, компонент, управляемый сообщениями, предоставляет мощный и простой компонент, специально предназначенный для получения сообщений JMS. В этой книге не рекомендуется пытаться получать сообщения в объектных или сеансовых компонентах.

Дополнительная информация о JMS

JMS (и корпоративные системы сообщений вообще) представляет собой мощную парадигму распределенных вычислений. По моему мнению, служба сообщений Java так же важна, как и Enterprise JavaBeans, и должна быть хорошо изучена перед применением ее в процессе разработки.

В этой главе приведен краткий обзор JMS, и она содержит лишь самый необходимый материал, призванный подготовить читателя к обсуждению компонентов, управляемых сообщениями, в следующем разделе. JMS имеет множество особенностей и деталей, которые просто слишком обширны, чтобы рассмотреть их все в этой книге. Чтобы понять

JMS и особенности ее применения, необходимо изучать ее самостоятельно.[1] Время, потраченное на изучение JMS, окупится с лихвой.

Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями (Message-Driven Beans, MDB), – это не имеющие состояния, серверные компоненты для обработки асинхронных сообщений JMS, поддерживающие транзакции. Впервые представленные в EJB 2.0, компоненты, управляемые сообщениями, обрабатывают сообщения, распространяемые через службу сообщений Java (Java Message Service).

Компоненты, управляемые сообщениями, могут принимать сообщения JMS и обрабатывать их. Компонент, управляемый сообщениями, отвечает за обработку сообщений, а его контейнер заботится об автоматическом управлении всем окружением компонента, включающим в себя транзакции, безопасность, ресурсы, совместный доступ и подтверждения получения сообщений.

Один из наиболее важных аспектов компонентов, управляемых сообщениями, состоит в том, что они могут получать и обрабатывать сообщения параллельно. Эта возможность дает им значительное преимущество перед обычными клиентами JMS, в которых управление ресурсами, транзакциями и безопасностью в многопоточном окружении должно выполняться самим клиентом. Контейнеры компонентов, управляемых сообщениями, поставляемые вместе с EJB, управляют совместным доступом автоматически, поэтому разработчик компонента может направить свои усилия только на создание прикладной логики обработки сообщений. MDB может принимать сотни сообщений JMS от разных прикладных программ и обрабатывать их все в одно и то же время благодаря тому, что в контейнере могут выполняться несколько экземпляров MDB одновременно.

Компонент, управляемый сообщениями, – это полноценный компонент, точно такой же, как и сеансовый или объектный компонент, но имеющий несколько важных отличий. Хотя у компонента, управляемого сообщениями, есть и класс компонента и XML-дескриптор развертывания, у него нет компонентных интерфейсов. Компонентные интерфейсы отсутствуют из-за того, что компонент, управляемый сообщениями, не доступен через программный интерфейс Java RMI, он отвечает только на асинхронные сообщения.

---

[1] Детали работы JMS описываются в «Java Message Service» (Служба сообщений Java) Ричарда Монзона-Хейфела (Richard Monson-Haefel) и Дэвида Чапела (David Chappell), издательство O'Reilly.

Компонент ReservationProcessor

Компонент ReservationProcessor – это компонент, управляемый сообщениями, который принимает сообщения JMS, информирующие его о новых заказах билетов. Компонент ReservationProcessor представляет собой автоматизированную версию компонента TravelAgent, обрабатывающую заказы билетов, переданные через JMS другими туристическими организациями. Он не требует никакого вмешательства человека, он полностью автоматизирован.

Сообщения JMS, уведомляющие компонент ReservationProcessor о новых заказах билетов, могли бы исходить из другого приложения предприятия или приложения, находящегося в другой организации. Когда компонент ReservationProcessor принимает сообщение, он создает новый компонент Reservation (добавляя его к базе данных), с помощью компонента ProcessPayment обрабатывает оплату и подготавливает билет. Этот процесс показан на рис. 13.3.

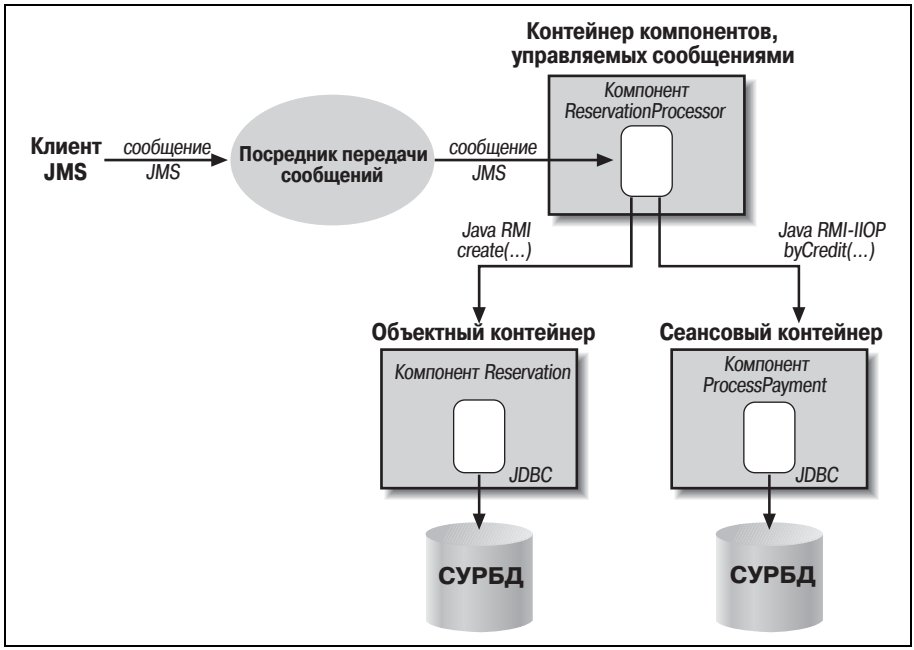

Рис. 13.3. Компонент ReservationProcessor, обрабатывающий заказы билетов

Класс ReservationProcessorBean

Здесь приведен фрагмент определения класса `ReservationProcessorBean`. Некоторые методы оставлены пустыми, они будут заполнены позже. Обратите внимание, что метод `onMessage()` содержит прикладную логику класса компонента, похожую на прикладную логику, содержащуюся в методе `bookPassage()` компонента TravelAgent в главе 12:

```
package com.titan.reservationprocessor;

import javax.jms.Message;
import javax.jms.MapMessage;
import com.titan.customer.*;
import com.titan.cruise.*;
import com.titan.cabin.*;
import com.titan.reservation.*;
import com.titan.processpayment.*;
import com.titan.travelagent.*;
import java.util.Date;

public class ReservationProcessorBean implements
javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext mdc) {
        ejbContext = mdc;
        try {
            jndiContext = new InitialContext();
        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }

    public void ejbCreate() {}

    public void onMessage(Message message) {
        try {
            MapMessage reservationMsg = (MapMessage)message;

            Integer customerPk =
                 (Integer)reservationMsg.getObject("CustomerID");
            Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
            Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");

            double price = reservationMsg.getDouble("Price");

            // Получаем кредитную карточку
            Date expirationDate =
                new Date(reservationMsg.getLong("CreditCardExpDate"));
            String cardNumber = reservationMsg.getString("CreditCardNum");
            String cardType = reservationMsg.getString("CreditCardType");
            CreditCardDO card = new CreditCardDO(cardNumber,
                expirationDate, cardType);

            CustomerRemote customer = getCustomer(customerPk);
            CruiseLocal cruise = getCruise(cruisePk);
            CabinLocal cabin = getCabin(cabinPk);

            ReservationHomeLocal resHome = (ReservationHomeLocal)
              jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");
```

```
            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price, new Date());

            Object ref = jndiContext.lookup
                ("java:comp/env/ejb/ProcessPaymentHomeRemote");

            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

            deliverTicket(reservationMsg, ticket);

        } catch(Exception e) {
            throw new EJBException(e);
        }
    }

    public void deliverTicket(MapMessage reservationMsg, TicketDO ticket) {

        // Посылаем его в нужный пункт назначения
    }
    public CustomerRemote getCustomer(Integer key)
        throws NamingException, RemoteException, FinderException {
        // Получаем удаленную ссылку на компонент Customer
    }
    public CruiseLocal getCruise(Integer key)
        throws NamingException, FinderException {
        // Получаем локальную ссылку на компонент Cruise
    }
    public CabinLocal getCabin(Integer key)
        throws NamingException, FinderException {
        // Получаем локальную ссылку на компонент Cabin
    }

    public void ejbRemove() {
        try {
            jndiContext.close();
            ejbContext = null;
        } catch(NamingException ne) { /* ничего не делаем */ }
    }
}
```

Интерфейс MessageDrivenBean

Классу компонента, управляемого сообщениями, необходимо реализовать интерфейс `javax.ejb.MessageDrivenBean`, в котором определены методы обратного вызова, похожие на те, которые были определены для объектных и сеансовых компонентов. Приведем определение интерфейса `MessageDrivenBean`:

```
package javax.ejb;

public interface MessageDrivenBean extends javax.ejb.EnterpriseBean {
    public void setMessageDrivenContext(MessageDrivenContext context)
        throws EJBException;
    public void ejbRemove() throws EJBException;
}
```

Метод `setMessageDrivenContext()` вызывается в начале жизненного цикла MDB и предоставляет экземпляру MDB ссылку на его `MessageDrivenContext`:

```
MessageDrivenContext ejbContext;
Context jndiContext;

public void setMessageDrivenContext(MessageDrivenContext mdc) {
    ejbContext = mdc;
    try {
        jndiContext = new InitialContext();
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

Метод `setMessageDrivenContext()` класса `ReservationProcessorBean` устанавливает поле экземпляра `ejbContext` в значение `MessageDrivenContext`, которое было передано в метод. Кроме этого он получает ссылку на JNDI ENC, которую сохраняет в `jndiContext`. MDB может иметь поля экземпляра, похожие на поля экземпляра сеансового компонента без состояния. Значения этих полей сохраняются в экземпляре MDB в течение всей его жизни и могут многократно использоваться каждый раз, когда он обрабатывает новое сообщение. В отличие от сеансовых компонентов с состоянием, у MDB нет состояния диалога, и они не предназначаются для одного клиента JMS. Экземпляры MDB применяются для обработки сообщений от нескольких разных клиентов JMS и связаны не с клиентом, а с определенной темой или очередью, от которых они принимают сообщения. Как и у сеансовых компонентов без состояния, у них нет состояния.

Метод `ejbRemove()` предоставляет экземпляру MDB возможность освобождения всех ресурсов, которые он содержит в своих полях экземпляра. В нашем случае он применяется для закрытия контекста JNDI и установки поля `ejbContext` в `null`. Эти действия не являются абсолютно необходимыми, но они показывают, что может происходить в методе `ejbRemove()`. Обратите внимание, что `ejbRemove()` вызывается в конце жизненного цикла MDB перед тем, как он будет удален сборщиком мусора. Он не может быть вызван в случаях, когда сервер EJB, выполняющий данный MDB, терпит крах или когда в одном из методов экземпляра MDB генерируется исключение `EJBException`. Когда каким-либо методом экземпляра MDB возбуждается исключение `EJBException` (или

любой тип RuntimeException), экземпляр немедленно удаляется из памяти, а транзакция откатывается назад.

MessageDrivenContext

MessageDrivenContext просто расширяет EJBContext, он не содержит никаких новых методов. EJBContext определен как:

```
package javax.ejb;
public interface EJBContext {

    // Транзакционные методы
    public javax.transaction.UserTransaction getUserTransaction()
        throws java.lang.IllegalStateException;
    public boolean getRollbackOnly() throws java.lang.IllegalStateException;
    public void setRollbackOnly() throws java.lang.IllegalStateException;

    // Внутренние методы компонента
    public EJBHome getEJBHome();
    public EJBLocalHome getEJBLocalHome();

    // Методы безопасности
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String roleName);

    // Отмененные методы
    public java.security.Identity getCallerIdentity();
    public boolean isCallerInRole(java.security.Identity role);
    public java.util.Properties getEnvironment();

}
```

Компонентам, управляемым сообщениями, доступны только те транзакционные методы, которые MessageDrivenContext наследует от EJBContext. Внутренние методы – getEJBHome() и getEJBLocalHome() – при вызове генерируют исключение RuntimeException, т. к. у MDB нет ни внутренних интерфейсов, ни внутренних объектов. Методы безопасности интерфейса MessageDrivenContext – getCallerPrincipal() и isCallerInRole() – также при вызове генерируют RuntimeException. Когда MDB обслуживает сообщение JMS, не существует никакого «вызывающего объекта», поэтому отсутствует какой-либо контекст безопасности, который можно было бы получить из него. Помните, что служба JMS является асинхронной и не передает получателю контекст безопасности отправителя. Это не имело бы смысла, поскольку отправители и получатели обычно работают в разных окружениях.

MDB обычно выполняются в инициированной контейнером или компонентом транзакции, поэтому транзакционные методы позволяют MDB управлять своим контекстом. Контекст транзакции не передается от отправителя JMS, а инициируется либо контейнером, либо компонентом явным использованием javax.jta.UserTransaction. Транзакционные методы EJBContext обсуждаются более подробно в главе 14.

Кроме этого компоненты, управляемые сообщениями, имеют доступ к своему собственному контексту имен окружения (Environment Naming Context, ENC) JNDI, который предоставляет экземплярам MDB доступ к элементам окружения, к другим компонентам и ресурсам. Например, компонент ReservationProcessor пользуется возможностями JNDI ENC для получения ссылок на компоненты Customer, Cruise, Cabin, Reservation и ProcessPayment, а также `QueueConnectionFactory` JMS и `Queue` для рассылки билетов.

Интерфейс MessageListener

Кроме интерфейса `MessageDrivenBean` MDB реализуют интерфейс `javax.jms.MessageListener`, в котором определен метод `onMessage()`. Разработчики компонентов реализуют данный метод для обработки сообщений JMS, получаемых компонентом. Именно в методе `onMessage()` компонент обрабатывает сообщение JMS:

```
package javax.jms;
public interface MessageListener {
    public void onMessage(Message message);
}
```

Интересно рассмотреть, почему MDB реализует интерфейс `MessageListener` отдельно от интерфейса `MessageDrivenBean`. Почему бы просто не поместить метод `onMessage()`, единственный метод интерфейса `MessageListener`, в интерфейс `MessageDrivenBean` так, чтобы в классе MDB надо было реализовывать только один интерфейс? Именно это решение было принято в ранней рабочей версии EJB 2.0. Однако разработчики быстро осознали, что в будущем компоненты, управляемые сообщениями, могли бы обрабатывать сообщения из других типов систем, а не только JMS. Для того чтобы сделать MDB открытым для других систем сообщений, было решено, что он должен реализовать интерфейс `javax.jms.MessageListener` отдельно, таким образом отделяя понятие компонента, управляемого сообщениями, от типа сообщений, которые он может обрабатывать. В будущих версиях спецификации могут стать доступными другие типы MDB для таких технологий, как SMTP (электронная почта) или JAXM (Java API для XML-сообщений) для ebXML. Эти технологии будут использовать методы, отличные от `onMessage()`, являющегося специфичным для JMS.

Метод OnMessage (): Рабочий поток и интеграция с B2B

Именно в методе `onMessage()` выполняется вся прикладная логика. Сообщения по мере их поступления передаются в MDB контейнером через метод `onMessage()`. После возвращения из метода MDB готов обработать новое сообщение.

В компоненте ReservationProcessor метод `onMessage()` извлекает информацию, относящуюся к заказу билетов, из `MapMessage` и использует эту информацию для выполнения резервирования в системе:

```
public void onMessage(Message message) {
    try {
        MapMessage reservationMsg = (MapMessage)message;

        Integer customerPk = (Integer)reservationMsg.getObject("CustomerID");
        Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
        Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");

        double price = reservationMsg.getDouble("Price");

        // Получаем кредитную карточку

        Date expirationDate =
            new Date(reservationMsg.getLong("CreditCardExpDate"));
        String cardNumber = reservationMsg.getString("CreditCardNum");
        String cardType = reservationMsg.setString("CreditCardType");
        CreditCardDO card = new CreditCardDO(cardNumber,
            expirationDate, cardType);
```

JMS нередко используется в качестве точки интеграции для прикладных программ типа «бизнес-бизнес», поэтому легко представить себе сообщение о заказе билетов, исходящее от одного из деловых партнеров системы «Титан» (возможно, от посредника или отраслевого транспортного агентства).

Для того чтобы обработать заказ билетов, компоненту ReservationProcessor необходимо обратиться к компонентам Customer, Cruise и Cabin. MapMessage содержит первичные ключи для этих объектов. Компонент ReservationProcessor применяет вспомогательные методы (`getCustomer()`**,** `getCruise()` **и** `getCabin()`**) для поиска объектных компонентов и получения на них ссылок компонентных объектов:**

```
public void onMessage(Message message) {
    ...
    CustomerRemote customer = getCustomer(customerPk);
    CruiseLocal cruise = getCruise(cruisePk);
    CabinLocal cabin = getCabin(cabinPk);
    ...
}

public CustomerRemote getCustomer(Integer key)
    throws NamingException, RemoteException, FinderException {

    Object ref = jndiContext.lookup("java:comp/env/ejb/CustomerHomeRemote");
    CustomerHomeRemote home = (CustomerHomeRemote)
        PortableRemoteObject.narrow(ref, CustomerHomeRemote.class);
    CustomerRemote customer = home.findByPrimaryKey(key);
    return customer;
}
public CruiseLocal getCruise(Integer key)
    throws NamingException, FinderException {

    CruiseHomeLocal home = (CruiseHomeLocal)
```

```
            jndiContext.lookup("java:comp/env/ejb/CruiseHomeLocal");
        CruiseLocal cruise = home.findByPrimaryKey(key);
        return cruise;
    }
    public CabinLocal getCabin(Integer key)
        throws NamingException, FinderException{

        CabinHomeLocal home = (CabinHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");
        CabinLocal cabin = home.findByPrimaryKey(key);
        return cabin;
    }
```

Информация, извлеченная из `MapMessage`, применяется для создания резервирования и обработки оплаты. Это в основном тот же рабочий поток, который использовался компонентом TravelAgent в главе 12. Компонент Reservation создается, чтобы представлять сам заказ билетов, а компонент ProcessPayment создается для обработки оплаты по кредитной карточке:

```
ReservationHomeLocal resHome = (ReservationHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

ReservationLocal reservation =
    resHome.create(customer, cruise, cabin, price, new Date());

Object ref = jndiContext.lookup("java:comp/env/ejb/
ProcessPaymentHomeRemote");

ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
    PortableRemoteObject.narrow (ref, ProcessPaymentHomeRemote.class);

ProcessPaymentRemote process = ppHome.create();
process.byCredit(customer, card, price);

TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

deliverTicket(reservationMsg, ticket);
```

Из этого видно, что, как и сеансовый компонент, MDB может обращаться к любому другому объектному или сеансовому компоненту и использовать его для выполнения своей задачи. Таким образом, MDB играет роль точки интеграции в сценариях с приложениями B2B. MDB может управлять процессом и взаимодействовать с другими компонентами и ресурсами. Например, для MDB обычной практикой является применение JDBC для обращения к базе данных на основе содержания сообщения, которое он обрабатывает.

Посылка сообщений из компонента, управляемого сообщениями

MDB также может посылать сообщения, используя JMS. Метод `deliverTicket()` посылает информацию о билете в пункт назначения, полученный из посылки клиента JMS:

```
public void deliverTicket(MapMessage reservationMsg, TicketDO ticket)
    throws NamingException, JMSException{

    Queue queue = (Queue)reservationMsg.getJMSReplyTo();

    QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");

    QueueConnection connect = factory.createQueueConneciton();

    QueueSession session = connect.createQueueSession(true,0);

    QueueSender sender = session.createSender(queue);

    ObjectMessage message = session.createObjectMessage();
    message.setObject(ticket);

    sender.send(message);

    connect.close();

}
```

Как было сказано выше, каждый тип сообщения состоит из двух частей: заголовка сообщения и тела сообщения (т. е. полезной информации). Заголовок сообщения содержит информацию о маршрутизации и может также содержать свойства для фильтрации сообщений и другие атрибуты, в том числе атрибут `JMSReplyTo`. Клиент JMS, посылая сообщение, может установить атрибут `JMSReplyTo` в значение какого-либо пункта назначения, доступного для его провайдера JMS.[1] В случае с сообщением о заказе билетов отправитель устанавливает атрибут `JMSReplyTo` в значение очереди, в которую должен быть послан итоговый билет. Другое приложение может обратиться к этой очереди, чтобы считать билеты и распределять их по клиентам или сохранить эту информацию в базе данных отправителя.

Можно также с помощью адреса `JMSReplyTo` указать на прикладную ошибку, произошедшую во время обработки сообщения. Например, если каюта уже занята, компонент ReservationProcessor мог бы послать в очередь `JMSReplyTo` сообщение об ошибке, объясняющее, что заказ не может быть обработан. Реализация такой обработки ошибок оставлена в качестве упражнения для читателя.

XML-дескриптор развертывания

У MDB есть XML-дескрипторы развертывания, точно так же, как у объектных и сеансовых компонентов. Они могут быть развернуты отдельно или, чаще всего, вместе с другими компонентами. Например,

[1] Очевидно, что если пункт назначения, указанный атрибутом JMSReplyTo, имеет тип Queue, должна быть использована модель сообщений «точка-точка» (на основе очередей). Если же пункт назначения имеет тип Topic, следует выбрать модель сообщений типа «издание-подписка» (на основе тем).

компонент ReservationProcessor необходимо было бы развертывать в том же архиве JAR, используя тот же XML-дескриптор развертывания, что и компоненты Customer, Cruise и Cabin, если он планирует использовать их локальные интерфейсы.

Здесь приведен XML-дескриптор развертывания, в котором определяется компонент ReservationProcessor. Этот дескриптор развертывания также определяет компоненты Customer, Cruise, Cabin и другие, но для краткости они опущены:

```
<enterprise-beans>
    ...
    <message-driven>
        <ejb-name>ReservationProcessorEJB</ejb-name>
        <ejb-class>
            com.titan.reservationprocessor.ReservationProcessorBean
        </ejb-class>
        <transaction-type>Container</transaction-type>
        <message-selector>MessageFormat = 'Version 3.4'</message-selector>
        <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
        <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
        </message-driven-destination>
        <ejb-ref>
            <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
            <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
            <local>com.titan.cruise.CruiseLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
          <local-home>com.titan.reservation.ReservationHomeLocal</local-home>
```

```
            <local>com.titan.reservation.ReservationLocal</local>
        </ejb-local-ref>
        <security-identity>
            <run-as>
                <role-name>everyone</role-name>
            </run-as>
        </security-identity>
        <resource-ref>
            <res-ref-name>jms/QueueFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </message-driven>
    ...
</enterprise-beans>
```

MDB объявлен в элементе `<message-driven>` внутри элемента `<enterprise-beans>` рядом с компонентами `<session>` и `<entity>`. Как и компоненты типа `<session>`, он определяет `<ejb-name>`, `<ejb-class>` и `<transaction-type>`, но он не определяет компонентные интерфейсы (ни локальный, ни удаленный). У MDB нет компонентных интерфейсов, вот почему эти определения не нужны.

<message-selector>

В MDB также может быть указан элемент `<message-selector>` (селектор сообщений), являющийся уникальным для компонентов, управляемых сообщениями:

```
<message-selector>MessageFormat = 'Version 3.4'</message-selector>
```

Селекторы сообщений позволяют MDB выбирать сообщения, которые он принимает из определенной темы или очереди. Свойства `Message` выступают как критерии в условных выражениях селекторов сообщений.[1] В этих условных выражениях для описания сообщений, которые должны быть доставлены клиенту, применяется булева логика.

Свойства сообщений, которые используются селектором сообщений, представляют собой дополнительные заголовки, которые могут быть связаны с сообщением. Они дают разработчику приложений или производителю JMS возможность присоединить к сообщению дополнительную информацию. Интерфейс Message предоставляет несколько методов для получения и изменения свойств. Свойства могут иметь значение `String` или одно из примитивных значений (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`). Система именования свойств, вместе с их значениями и правилами преобразования, строго определяется в JMS.

[1] Селекторы сообщений также могут использовать информацию из заголовков сообщений, но это выходит за рамки данной книги.

Компонент ReservationProcessor использует фильтр селектора сообщений для выбора сообщений определенного формата. В нашем случае форматом является строка «Version 3.4». По этой строке в системе «Титан» осуществляется идентификация сообщений типа MapMessage, содержащих значения имен CruiseID, CabinID, CreditCard и Price. Другими словами, указывая MessageFormat в каждом сообщении о резервировании, мы можем написать MDB, предназначенные для обработки разных видов сообщений о заказах билетов. Если новому деловому партнеру потребуется работать с отдельным типом объекта Message, «Титан» мог бы использовать новую версию сообщения и MDB для его обработки.

Ниже показано, как производитель JMS должен устанавливать свойство MessageFormat объекта Message:

```
Message message = session.createMapMessage();
message.setStringPropery("MessageFormat","Version 3.4");

// устанавливаем именованные значения для резервирования

sender.send(message);
```

Селекторы сообщений основаны на подмножестве синтаксиса условных выражений SQL-92, который используется в секции WHERE операторов SQL. Они могут быть достаточно сложными, включая применение литеральных значений, выражений типа Boolean, унарных операторов и т. д.

Примеры селекторов сообщений

Здесь приведены три сложных селектора, применяемые в гипотетических окружениях. И хотя вам придется немного напрячь воображение, цель этих примеров состоит в том, чтобы показать мощность селекторов сообщений. Во время объявления селектора идентификатор всегда ссылается на имя свойства или имя заголовка JMS. Например, селектор UserName !='William' предполагает, что в сообщении существует свойство с именем UserName, которое можно сравнивать со значением 'William'.

Управление заявками в HMO (Health Maintenance Organization). Из-за нескольких ложных заявок реализован автоматический процесс, использующий MDB, который будет отслеживать все заявки, направляемые пациентами (служащими производственной компании ACME), на посещение хиромантов, физиологов и дерматологов:

```
<message-selector>
<![CDATA[
    PhysicianType IN ('Chiropractic','Psychologists','Dermatologist')
    AND PatientGroupID LIKE 'ACME%'
]]>
</message-selector>
```

Операторы MDB <message-selector> объявляются в XML-дескрипторах развертывания. XML присваивает специальное значение ряду символов, таким как символы больше (>) и меньше (<), поэтому применение этих символов в операторах <message-selector> вызовет ошибки синтаксического анализа, если не использовать разделы CDATA. Разделы CDATA необходимы по той же причине, что и в операторах EJB QL <ejb-ql>, как было показано в главе 8.

Уведомление об определенных предложениях на товары. Продавец желает получать уведомления о запросах по предложению цены на определенный товар с заданным количеством:

```
<message-selector>
<![CDATA[
    InventoryID ='S93740283-02' AND Quantity BETWEEN 1000 AND 13000
]]>
</message-selector>
```

Выбор получателей для рассылки по каталогу. Интернет-продавец хочет разослать специальный каталог всем клиентам, сделавшим заказы на сумму, превышающую $500, если средняя стоимость заказанного товара превышает $75 и покупатель постоянно находится в одном из нескольких штатов. Продавец создает MBD, который подписывается на тему обработки заказов и обрабатывает доставку каталога только для тех клиентов, которые удовлетворяют заданным критериям:

```
<message-selector>
<![CDATA[
    TotalCharge >500.00 AND ((TotalCharge /ItemCount)>=75.00)
    AND State IN ('MN','WI','MI','OH')
]]>
</message-selector>
```

<acknowledge-mode>

В JMS есть понятие подтверждения (acknowledge[ment]), которое означает, что клиент JMS уведомляет провайдер JMS (маршрутизатор сообщений) о получении сообщения. В EJB посылка подтверждения провайдеру JMS является обязанностью контейнера MDB, когда он принимает сообщение. Подтверждение сообщения информирует провайдер JMS, что контейнер MDB принял сообщение и обработал его с помощью экземпляра MDB. Без подтверждения провайдер JMS не будет знать, принял ли контейнер MDB сообщение, поэтому он попытается послать его повторно. Это может вызывать проблемы. Например, после того как мы обработали сообщение о заказе билетов, используя компонент ReservationProcessor, мы не должны снова получать это же сообщение.

Когда мы имеем дело с транзакциями, режим подтверждения, установленный провайдером компонента, игнорируется. В этом случае подтверждение выполняется внутри контекста транзакции. Если транзакция завершается успешно, сообщение подтверждается. Если происходит сбой транзакции, сообщение не подтверждается. Если MDB основывается на управляемых контейнером транзакциях, как это будет происходить в большинстве случаев, то контейнер MDB игнорирует режим подтверждения. При использовании управляемых контейнером транзакций с атрибутом транзакции Required элемент `<acknowledge-mode>` обычно не указывается. Однако мы включили его в дескриптор развертывания в учебных целях:

```
<acknowledge-mode>Auto-acknowledge</acknowledge-mode>
```

Если MDB выполняется внутри управляемой компонентом транзакции или внутри управляемой контейнером транзакции с атрибутом `NotSupported` (глава 14), значение `<acknowledge-mode>` становится значимым.

Для `<acknowledge-mode>` могут быть заданы два значения: `Auto-acknowledge` и `Dups-ok-acknowledge`. `Auto-acknowledge` указывает контейнеру, что он должен посылать подтверждение провайдеру JMS сразу после того, как сообщение будет передано для обработки экземпляру MDB. `Dups-ok-acknowledge` указывает контейнеру, что он не должен посылать подтверждение немедленно, это может быть сделано в любое время после того, как сообщение будет передано экземпляру MDB. Если задано значение `Dups-ok-acknowledge`, становится возможной ситуация, когда контейнер MDB задержит подтверждение настолько, что провайдер JMS решит, что сообщение не было получено, и пошлет «продублированное» сообщение. Очевидно, что в случае с `Dups-ok-acknowledge` ваши MDB должны уметь правильно обрабатывать повторные сообщения.

`Auto-acknowledge` предупреждает появление повторных сообщений из-за того, что подтверждение посылается немедленно. Поэтому провайдер JMS не будет посылать дубликат. В большинстве MDB `Auto-acknowledge` применяются для того, чтобы избежать повторной обработки одного сообщения. `Dups-ok-acknowledge` существует потому, что он дает возможность провайдеру JMS оптимизировать использование сети. Тем не менее, на практике накладные расходы на подтверждения настолько малы, а частота передачи между контейнером MDB и провайдером JMS настолько высока, что `Dups-ok-acknowledge` не оказывает на эффективность большого влияния.

<message-driven-destination>

Элемент `<message-driven-destination>` определяет тип пункта назначения, из которого MDB принимает сообщения. Разрешенные значения для этого элемента – `javax.jms.Queue` и `javax.jms.Topic`. В компоненте ReservationProcessor это значение установлено в `javax.jms.Queue`, показывая, что MDB получает свои сообщения через модель передачи сообщений p2p из очереди:

```
<message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
</message-driven-destination>
```

Во время развертывания MDB необходимо связать его так, чтобы он прослушивал в сети существующую очередь.

Когда `<destination-type>` содержит `javax.jms.Topic`, элемент `<subscription-durability>` должен содержать в качестве своего значения или `Durable`, или `NonDurable`:

```
<message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
    <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

Элемент `<subscription-durability>` определяет, действительно ли подписка MDB на тему будет длительной (`Durable`). Длительная подписка «переживает» подключение контейнера MDB к провайдеру JMS. Так, если сервер EJB терпит частичный крах, выключается или как-то иначе отсоединяется от провайдера JMS, сообщения, которые он должен принять, не будут потеряны. Пока «длительный» контейнер MDB отсоединен от провайдера JMS, в обязанности провайдера входит сохранение всех сообщений, подписчик которых отсутствует. Когда длительный контейнер MDB воссоединяется с провайдером JMS, провайдер посылает ему все накопившиеся за время отсутствия связи не устаревшие сообщения. Такое поведение обычно называется *передачей сообщений с промежуточным накоплением* (*store-and-forward messaging*). Длительный MDB допускает потери соединения, являются ли они преднамеренными или происходят в результате частичного отказа.

Если `<subscription-durability>` установлен в `NonDurable`, все сообщения, которые компонент мог бы принять, пока он был отсоединен, будут потеряны. Разработчики используют подписки типа `NonDurable` в случаях, когда обработка всех сообщений не обязательна. Подписка `NonDurable` улучшает производительность провайдера JMS, но, вместе с тем, значительно уменьшает надежность MDB.

Если `<destination-type>` установлен в `javax.jms.Queue`, как в случае с компонентом ReservationProcessor, длительность не является критичной из-за самой природы p2p или систем сообщений, основанных на очереди. В случае применения очереди сообщения могут быть получены только один раз, и они могут оставаться в очереди, пока не будут переданы одному из слушателей очереди.

С остальными элементами дескриптора развертывания вы должны быть знакомы. Элемент `<ejb-ref>` предоставляет связь JNDI ENC для удаленного внутреннего объекта, тогда как элементы `<ejb-local-ref>` предоставляют связь JNDI ENC для локальных внутренних объектов. Обратите внимание, что элемент `<resource-ref>`, определяющий JMS `QueueConnectionFactory`, используемую компонентом ReservationProcessor

для отправки сообщения о билете, не соответствует никакому элементу <resource-env-ref>. Очередь, в которую посылаются билеты, извлекается из заголовка JMSReplyTo самой MapMessage, а не из JNDI ENC.

Клиенты ReservationProcessor

Для того чтобы протестировать компонент ReservationProcessor, нам необходимо разработать два новых клиентских приложения: одно – для отправки сообщения о заказе билетов, а второе – для приема сообщения о билете, сгенерированного компонентом ReservationProcessor.

Поставщик сообщений о заказе билетов. Компонент JmsClient_ReservationProducer предназначен для очень быстрой отправки 100 запросов по заказам билетов. Быстродействие, с которым он посылает эти сообщения, вынудит большинство контейнеров MDB использовать несколько экземпляров компонента для обработки сообщений о резервировании. Код для JmsClient_ReservationProducer выглядит следующим образом:

```
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.Queue;
import javax.jms.QueueSender;
import javax.jms.JMSException;
import javax.naming.InitalContext;
import java.util.Date;

import com.titan.processpayment.CreditCardDO;

public class JmsClient_ReservationProducer {

    public static void main(String [] args) throws Exception {

        InitialContext jndiContext = getInitialContext();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("ИмяФабрикиОчереди");

        Queue reservationQueue = (Queue)
            jndiContext.lookup("ИмяОчереди");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
            connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueSender sender = session.createSender(reservationQueue);

        Integer cruiseID = new Integer(1);

        for(int i = 0; i < 100; i++){
```

```
            MapMessage message = session.createMapMessage();
            message.setStringProperty("MessageFormat","Version 3.4");

            message.setInt("CruiseID",1);
            message.setInt("CustomerID",i%10);
            message.setInt("CabinID",i);
            message.setDouble("Price", (double)1000+i);

            // действие карточки истекает примерно через 30 дней
            Date expirationDate = new
            Date(System.currentTimeMillis()+43200000);
            message.setString("CreditCardNum", "923830283029");
            message.setLong("CreditCardExpDate", expirationDate.getTime());
            message.setString("CreditCardType", CreditCardDO.MASTER_CARD);

            sender.send(message);

        }

        connect.close();
    }

    public static InitialContext getInitialContext()
        throws JMSException {
        // Создаем контекст JNDI, специфичный для данного производителя
    }
}
```

Вы, возможно, заметили, что `JmsClient_ReservationProducer` **устанавливает** `CustomerID`**,** `CruiseID` **и** `CabinID` **как значения примитивного типа** `int`**, но** `ReservationProcessorBean` **считывает эти значения в виде типов** `java.lang.Integer`**. Это не ошибка.** `MapMessage` **автоматически преобразует все примитивные типы к их соответствующим оберткам, если эти примитивы считываются с помощью** `MapMessage.getObject()`**. Так, именованное значение, загружаемое в** `MapMessage` **с помощью** `setInt()`**, может быть с помощью** `getObject()` **считано в виде** `Integer`**. Например, следующий код устанавливает значение в виде примитива** `int`**, а затем обращается к нему как к объекту** `java.lang.Integer`**:**

```
MapMessage mapMsg = session.createMapMessage();

mapMsg.setInt("TheValue",3);

Integer myInteger = (Integer)mapMsg.getObject("TheValue");

if(myInteger.intValue() == 3 )
    // всегда true
```

Потребитель сообщения о билете. `JmsClient_TicketConsumer` **предназначен для получения всех сообщений о билетах, направляемых экземплярами компонента ReservationProcessor в очередь. Он получает сообщения и распечатывает их описания:**

```
import javax.jms.Message;
import javax.jms.ObjectMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.Queue;
import javax.jms.QueueReceiver;
import javax.jms.JMSException;
import javax.naming.InitalContext;

import com.titan.travelagent.TicketDO;

public class JmsClient_TicketConsumer
    implements javax.jms.MessageListener {

    public static void main(String [] args) throws Exception {

        new JmsClient_TicketConsumer();

        while(true){Thread.sleep(10000);}

    }

    public JmsClient_TicketConsumer() throws Exception {

        InitialContext jndiContext = getInitialContext();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("ИмяФабрикиОчереди");

        Queue ticketQueue = (Queue)jndiContext.lookup("ИмяОчереди");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
            connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueReceiver receiver = session.createReceiver(ticketQueue);

        receiver.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message) {
        try {

            ObjectMessage objMsg = (ObjectMessage)message;
            TicketDO ticket = (TicketDO)objMsg.getObject();
            System.out.println("*******************************");
            System.out.println(ticket);
            System.out.println("*******************************");

        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }
    public static InitialContext getInitialContext() throws JMSException {
```

```
            // создаем контекст JNDI, специфичный для производителя
        }
    }
```

Для того чтобы компонент ReservationProcessor мог работать с двумя клиентскими приложениями – `JmsClient_ReservationProducer` и `JmsClient_TicketConsumer`, вы должны настроить провайдер JMS вашего контейнера EJB так, чтобы у него было две очереди: одна для сообщений о резервировании, а другая – для сообщений о билетах.

📖 Рабочее упражнение 13.2. Компонент, управляемый сообщениями

Жизненный цикл компонента, управляемого сообщениями

Так же как объектные и сеансовые компоненты, компоненты MDB имеют четко определенный жизненный цикл. Жизненный цикл экземпляра MDB имеет два состояния: *«не существует»* и *«пул готовых методов»*. Пул готовых методов похож на пул экземпляров, используемый для сеансовых компонентов без состояния. Как и для компонентов без состояния, для жизненных циклов MDB определен пул экземпляров.[1]

На рис. 13.4 показаны состояния и переходы, через которые проходит экземпляр MDB на протяжении своего времени жизни.

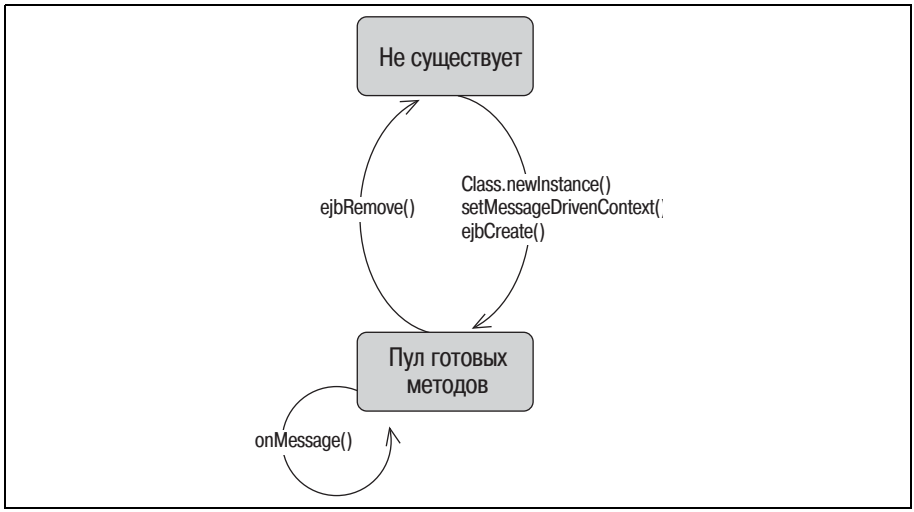

Рис. 13.4. Жизненный цикл MDB

[1] Некоторые производители могут не использовать пул для MDB-экземпляров, а вместо этого создавать и уничтожать экземпляры для каждого нового сообщения. Это нестандартное решение и оно не должно влиять на установленный жизненный цикл MDB-экземпляра компонента.

Не существует

Когда экземпляр MDB находится в состоянии «не существует», он не является экземпляром в памяти системы. Другими словами, он еще не создан.

Пул готовых методов

Экземпляры MDB переходят в пул готовых методов по мере того, как они становятся нужны контейнеру. При первом запуске сервер EJB может создавать несколько экземпляров MDB, помещая их в пул готовых методов. (Действительное поведение сервера зависит от реализации.) Когда количество экземпляров MDB, обрабатывающих входящие сообщения, становится недостаточным, могут быть созданы и добавлены к пулу дополнительные.

Переход в пул готовых методов

Когда экземпляр перемещается из состояния «не существует» в пул готовых методов, над ним выполняются три действия. Сначала посредством метода `Class.newInstance()` класса MDB создается экземпляр компонента. Далее контейнер вызывает метод `setMessageDrivenContext()`, предоставляя экземпляру MDB ссылку на его `EJBContext`. Ссылка на `MessageDrivenContext` может быть сохранена в поле экземпляра MDB.

Наконец, контейнером вызывается безаргументный метод `ejbCreate()` экземпляра компонента. У MDB есть только один метод `ejbCreate()`, не принимающий никаких параметров. Метод `ejbCreate()` вызывается только один раз в течение жизненного цикла MDB.

MDB не являются объектами, подлежащими активации, поэтому они могут поддерживать открытые соединения с ресурсами на всем протяжении их жизненного цикла.[1] Метод `ejbRemove()` должен закрывать все открытые ресурсы прежде, чем MDB будет удален из памяти в конце своего жизненного цикла.

Жизнь в пуле готовых методов

Если экземпляр находится в пуле готовых методов, он готов к обработке входящих сообщений. Сообщение, поступающее к MDB, перенаправляется любому доступному экземпляру, находящемуся в пуле готовых методов. Пока экземпляр выполняет запрос, он не может обрабатывать другие сообщения. MDB может одновременно обрабатывать

[1] Предполагается, что продолжительность жизни MDB-экземпляра будет очень большой. Однако в действительности некоторые серверы EJB могут уничтожать и создавать экземпляры для каждого нового сообщения, что делает такую тактику менее привлекательной. Подробности обработки MDB-экземпляров можно найти в документации по серверу.

несколько сообщений, делегируя обязанность по обработке каждого сообщения другому экземпляру MDB. Когда сообщение направлено экземпляру контейнером, `MessageDrivenContext` экземпляра MDB изменяется, чтобы отразить новый контекст транзакции. Экземпляр, закончивший обработку, сразу же становится доступным для обработки нового сообщения.

Переход из пула готовых методов: Смерть экземпляра MDB

Экземпляры компонентов переводятся из пула готовых методов в состояние «не существует», когда они становятся ненужными серверу. Это происходит, когда сервер принимает решение уменьшить общий размер пула готовых методов, удаляя из памяти один или несколько экземпляров. Этот процесс начинается с вызова метода `ejbRemove()` экземпляра. В это время экземпляр компонента должен выполнить все необходимые действия по очистке, такие как закрытие открытых ресурсов. Метод `ejbRemove()` вызывается только один раз за жизненный цикл экземпляра MDB, когда контейнер собирается переводить его в состояние «не существует». Во время выполнения метода `ejbRemove()` для экземпляра компонента все еще доступен контекст `MessageDrivenContext` и открыт доступ к JNDI ENC. По завершении метода `ejbRemove()` удаляются все указывающие на компонент ссылки, после чего он удаляется сборщиком мусора.

14

Транзакции

ACID-транзакции

Для того чтобы понять работу транзакций, еще раз посмотрим на компонент TravelAgent – сеансовый компонент с состоянием, инкапсулирующий процесс резервирования билетов на круиз для клиента.

В EJB 2.0 метод `bookPassage()` **компонента TravelAgent выглядит следующим образом:**

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");
        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);
        Object ref = jndiContext.lookup
            ("java:comp/env/ejb/ProcessPaymentHomeRemote");
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
           PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
```

```
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
```

В EJB 1.1 метод `bookPassage()` **выглядит так:**

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome = (ReservationHomeRemote)
            getHome("ReservationHomeRemote", ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            getHome("ProcessPaymentHomeRemote",
ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

Компонент TravelAgent представляет собою довольно простой сеансовый компонент, и использование им других компонентов – типичный пример проектирования прикладных объектов и рабочего потока. К сожалению, только хорошего дизайна прикладных объектов не достаточно для того, чтобы сделать эти компоненты полезными в мощных промышленных приложениях. Проблема состоит не в определении компонентов или рабочего потока; а в том, что хороший дизайн сам по себе не гарантирует того, что метод `bookPassage()` компонента TravelAgent представляет надежную *транзакцию* (*transaction*). Чтобы понять почему, посмотрим внимательно, что такое транзакция и каким критериям она должна удовлетворять, чтобы считаться надежной.

В коммерческих продуктах транзакция обычно имеет дело с обменом между двумя участниками. Покупая стаканчик мороженого, вы обмениваете деньги на продовольствие; работая на компанию, вы обмениваете свой опыт и время на деньги (на которые можно купить еще больше мороженого). Принимая участие в этих обменах, вы контролируете результат, чтобы быть уверенным, что вас не ограбят. Отдавая

продавцу мороженого деньги, вы не хотите, чтобы он ушел и не дал вам ничего взамен. Точно так же, вы хотите быть уверены, что ваша зарплата соответствует времени, проведенному вами на работе. Контролируя эти коммерческие обмены, вы пытаетесь гарантировать надежность транзакций. Вы убеждаетесь, что каждая транзакция удовлетворяет всех.

В коммерческом программном обеспечении транзакция воплощает понятие коммерческого обмена. Прикладная системная транзакция (или просто транзакция) представляет собой *единицу работы* (*unit-of-work*), которая обращается к одному или нескольким общим ресурсам, обычно базам данных. Единица работы – это набор связанных друг с другом действий, которые должны быть завершены вместе. Процесс резервирования представляет собой единицу работы, состоящую из нескольких действий: запись заказа, снятие денег с кредитной карточки и создание билета совместно составляют одну единицу работы.

Транзакции являются частью множества систем разных типов. Каждая транзакция преследует одну и ту же цель: выполнить единицу работы, которая завершилась бы надежным обменом.

Далее приведено несколько примеров других типов прикладных систем, использующих транзакции:

Банкомат (Automatic Teller Machine, ATM)

Банкомат, при помощи которого вы вносите, снимаете и переводите деньги, выполняет эти единицы работы в виде транзакций. При снятии денег, например, он проверяет, не превышен ли вами определенный лимит, а затем уменьшает ваш счет и выдает некоторое количество денег.

Заказ книг через Интернет

Возможно, многие из ваших книг по Java – возможно, даже эта книга – куплены через книжный интернет-магазин. Этот вид покупки также представляет собой единицу работы, имеющую вид транзакции. Покупая книги через Интернет, вы посылаете номер своей кредитной карты, он проверяется, и с карты снимается сумма, равная цене книги. После этого заказ на доставку книги отсылается на склад магазина.

Медицинская система

В медицинских системах каждый день регистрируются важные данные (некоторые из них – критические), относящиеся к пациентам. Это информация о посещениях врачей, медицинских процедурах, предписаниях и аллергии на препараты. Врач выписывает лекарство, затем система проверяет возможность аллергических реакций, противопоказания и необходимые дозировки. Если все проверки прошли успешно, препарат может быть назначен. Только что описанные задачи в медицинской системе представляют собой единицу работы. Единица работы в медицинской системе может быть не свя-

зана с финансами, но это не так важно. Сбой при определении аллергии пациента на какой-либо препарат может иметь фатальный исход.

Как видите, транзакции часто довольно сложны и обычно связаны с манипуляцией большими объемами данных. Ошибки в данных могут стоить денег или даже жизни. Поэтому транзакции должны гарантировать целостность данных, а это означает, что транзакция каждый раз должна выполняться либо успешно, либо не выполняться вообще. Это достаточно трудная задача, особенно для сложных систем. Каким бы трудным для выполнения ни было это требование, имея дело с коммерцией, мы не имеем права на ошибку. Единица работы, связанная с деньгами или чем-нибудь ценным, всегда требует предельной надежности, поскольку ошибки влияют на доходы и благосостояние заинтересованных сторон.

Чтобы получить представление о точности, требуемой транзакциями, подумайте о том, что может случиться, если в транзакционной системе произойдет случайная ошибка. Банкоматы дают клиентам удобный доступ к их банковским счетам и представляют значительную часть всех транзакций в персональном банковском деле. Транзакции, обрабатываемые банкоматами, просты, но многочисленны и дают нам убедительное обоснование необходимости защиты транзакций от ошибок. Скажем, у банка есть 100 банкоматов в городской зоне, и каждый обрабатывает 300 транзакций (вклады, изъятия или переводы) в день с общим количеством 30 000 транзакций в день. Если каждая транзакция, в среднем, связана с вкладом, изъятием или передачей приблизительно 100 долларов, то через систему банкоматов будет проходить приблизительно три миллиона долларов в день. Годовой оборот превысит миллиард долларов:

365 дней × 100 банкоматов × 300 транзакций × $100,00 =
= $1 095 000 000,00

Насколько хорошо должен работать банкомат, чтобы его работа считалась надежной? Допустим, что банкоматы выполняют транзакции правильно в 99,99% случаев. Это кажется более чем адекватным: в конце концов, лишь одна из каждых десяти тысяч транзакций выполняется неправильно. Но, произведя несложный подсчет, мы увидим, что за год ошибка составит более 100 000 долларов!

$1 095 000 000,00 × 0,01 % = $109 500,00

Очевидно, что мы сильно упростили задачу, но это показывает, что в крупномасштабных и высоконадежных системах не допускаются даже ничтожные ошибки. По этой причине эксперты выделили четыре характеристики транзакций, которым должна удовлетворять система, считающаяся надежной. Транзакции должны быть *атомарными*, *согласованными*, *изолированными* и *устойчивыми* (*atomic*, *consistent*, *isolated*, *durable* – ACID) – четыре столпа службы транзакций. Далее показано, что означает каждый из этих терминов.

Атомарность

Чтобы быть атомарной, транзакция должна или выполняться целиком, или не выполняться совсем. Это означает, что каждая задача в рамках единицы работы должна выполняться без ошибок. Если одна из задач завершится неудачно, вся единица работы, или транзакция, прерывается, а это значит, что все сделанные изменения данных отменяются. Если все задачи выполняются успешно, транзакция подтверждается, это означает, что изменения данных делаются постоянными (или устойчивыми).

Согласованность

Согласованность – это характеристика транзакции, которая должна гарантироваться и транзакционной системой, и разработчиком приложения. Согласованность связана с целостностью базовой памяти данных. Транзакционная система выполняет свои обязательства по отношению к согласованности, гарантируя, что транзакция является атомарной, изолированной и устойчивой. Разработчик приложения должен гарантировать, что база данных имеет подходящие ограничения (первичные ключи, ссылочная целостность и т. д.) и что единица работы (прикладная логика) не приведет к несогласованности данных (т. е. к данным, которые не соответствуют реальному миру, который они представляют). При переводе со счета, например, дебет одного счета должен равняться кредиту другого счета.

Изолированность

Транзакция должна иметь возможность выполниться без взаимного влияния других процессов или транзакций. Другими словами, на данные, к которым осуществляется доступ изнутри транзакции, не должны влиять никакие другие части системы до завершения транзакции, или единицы работы.

Устойчивость

Устойчивость означает, что все изменения данных, сделанные в течение транзакции, должны быть записаны в некоторый тип физической памяти, прежде чем транзакция успешно завершится. Это гарантирует, что при крахе системы изменения не будут потеряны.

Для того чтобы лучше представить, что означают эти принципы, рассмотрим компонент TravelAgent с точки зрения этих четырех свойств ACID.

Является ли компонент TravelAgent атомарным?

Наш первый признак надежности компонента TravelAgent – его атомарность. Гарантируется ли, что транзакция выполняется целиком или не выполняется совсем? С чем мы действительно имеем дело, так это с критическими задачами, изменяющими или создающими инфор-

мацию. В методе bookPassage() создается компонент Reservation, компонент ProcessPayment дебетует кредитную карточку и создается объект TicketDO. Для того чтобы вся транзакция была проведена успешно, каждая из этих задач должна быть успешно выполнена.

Для того чтобы понять важность атомарности, необходимо представить, что случится, если хотя бы одна из подзадач не сможет быть выполнена. Если, например, создание компонента Reservation потерпело неудачу, но все другие задачи завершились успешно, то пассажира могут исключить из списка участников круиза или поместить в одну каюту с незнакомцем. С точки зрения туристического агента метод bookPassage() был выполнен успешно, потому что TicketDO был создан. Если билет сгенерирован без создания предварительного заказа (брони), то состояние прикладной системы перестает соответствовать действительности, т. к. пассажир оплатил билет, но заказ не был зарегистрирован. Аналогично, если компонент ProcessPayment будет не в состоянии принять платеж с кредитной карточки пассажира, то последнему будет предоставлен бесплатный круиз. Он-то, может быть, и будет счастлив, но руководство – нет. Наконец, если TicketDO не будет создан, клиент не будет иметь записи в транзакции и, вероятно, не будет допущен на судно.

Поэтому единственный способ, которым может быть завершен метод bookPassage(), предполагает, что все критические операции завершились успешно. Если что-то идет не так, как надо, то должен быть прерван весь процесс. Прерывание выполнения транзакции требует не просто незавершения задач, более того, для всех задач, выполнявшихся внутри транзакции, должен быть выполнен откат. Если, например, создание компонента Reservation и вызов метода ProcessPayment.byCredit() прошло удачно, но создание TicketDO потерпело неудачу, сгенерировав исключение конструктора, то записи о резервировании и оплате не должны быть добавлены в базу данных.

Является ли компонент TravelAgent согласованным?

Для того чтобы транзакция была согласованной, прикладная система после завершения транзакции должна иметь смысл. Другими словами, *состояние* прикладной системы должно быть согласовано с реальной прикладной задачей. А для этого транзакция должна быть атомарной, изолированной и устойчивой, и, кроме того, необходимо надежное обеспечение ограничений целостности разработчиком приложений. Если, например, разработчик приложения не реализует в методе bookPassage() операцию, принимающую платежи с кредитной карточки, то пассажиру будет выдан билет, но с него не будет взята плата. С точки зрения бизнеса данные будут несогласованными, т. к. пассажир должен платить за проезд.

Кроме того, база данных должна быть настроена так, чтобы обеспечивать ограничения целостности. Например, не должно быть возможнос-

ти добавить записи к таблице RESERVATION, если внешние ключи CABIN_ID, CRUISE_ID и CUSTOMER_ID не соответствуют никаким записям в таблицах CABIN, CRUISE и CUSTOMER соответственно. Если используется CUSTOMER_ID, не сопоставленный с записью в таблице CUSTOMER, то ограничение ссылочной целостности должно вызвать генерацию базой данных сообщения об ошибке.

Является ли компонент TravelAgent изолированным?

Те, кто знаком с понятием синхронизации потоков в Java или схемами блокировки строк в реляционных базах данных, знаком и с понятием изоляции. Для того чтобы быть изолированной, транзакция должна защищать данные, к которым она обращается, от других транзакций. Необходимо предотвратить другие транзакции от взаимодействия с данными, находящимися в процессе изменения. В компоненте TravelAgent транзакция изолируется с целью предотвращения изменения модифицируемых компонентов другими транзакциями. Представьте себе, что отдельным транзакциям разрешено в любое время изменять любой объектный компонент. В этом случае транзакции могут наложиться друг на друга, а мы с легкостью можем получить несколько пассажиров, заказавших одну и ту же каюту, – из-за того, что их туристические агенты случайно сделали заказы билетов в одно и то же время.

Изоляция данных, к которым имеют доступ компоненты, не означает того, что на время выполнения транзакции происходит блокировка всего приложения. Изолируются только те компоненты и данные, на которые транзакция влияет непосредственно. В компоненте TravelAgent, например, транзакция изолирует только созданный компонент Reservation. Могут существовать несколько компонентов Reservation, и не существует причин, по которым к ним нельзя обращаться из других транзакций.

Является ли компонент TravelAgent устойчивым?

Устойчивость метода bookPassage() означает, что он считается успешным, если записал все сделанные изменения и новые данные в постоянную память данных. Хотя это кажется очевидным, зачастую в реальной жизни это не делается. Ради улучшения производительности изменения перед сохранением на диск часто хранятся в памяти в течение длительного времени. Смысл в том, чтобы уменьшить количество обращений к диску, тормозящих систему, и лишь периодически записывать накопленные изменения в базу данных. Этот подход превосходен с точки зрения эффективности, но он в то же время достаточно опасен, т. к. данные могут быть потеряны при выключении системы и очистке памяти. Устойчивость требует, чтобы система сохраняла все изменения, сделанные внутри транзакции, после успешного завершения последней, таким образом сохраняя целостность данных.

В случае компонента TravelAgent это означает, что вновь вставляемые записи RESERVATION и PAYMENT будут сделаны постоянными до того, как транзакция сможет успешно завершиться. Лишь когда данные сделаны устойчивыми, эти отдельные записи станут доступными из других транзакций через соответствующие им компоненты. Следовательно, устойчивость также играет свою роль в изоляции. Транзакция не считается законченной, пока данные не будут успешно записаны.

Для обеспечения соответствия транзакции принципам ACID необходимо тщательное проектирование. Система должна контролировать развитие транзакции, для того чтобы гарантировать, что она делает всю свою работу, что данные изменяются правильно, что транзакции не пересекаются друг с другом и что внесенные изменения не будут потеряны при аварийном отказе системы. Реализация всех этих возможностей системы требует большой работы, и никому не захочется повторять ее для каждой отдельной прикладной системы. К счастью, EJB обеспечивает автоматическую поддержку транзакций, упрощая процесс создания транзакционных систем. В оставшейся части этой главы рассматривается, как EJB поддерживает транзакции неявно (через декларативные атрибуты транзакций) и явно (через транзакционный API Java).

Декларативное управление транзакциями

Одно из основных достоинств Enterprise JavaBeans состоит в том, что она разрешает декларативное управление транзакциями. В противном случае транзакциями пришлось бы управлять с помощью явного указания границ транзакций. Это связано с применением достаточно сложных API, таких как службы объектных транзакций (Object Transaction Services, OTS) OMG или их Java-реализации – службы транзакций Java (Java Transaction Service, JTS). Явное указание границ транзакций представляет определенную трудность для разработчиков, особенно для новичков в системах транзакций. Кроме того, явное установление границ транзакции требует, чтобы код, управляющий транзакциями, был включен в прикладную логику, что уменьшает понятность кода и, что более важно, создает негибкие распределенные объекты. А если границы транзакций будут жестко закодированы в прикладном объекте, то для изменения поведения транзакции потребуются изменения непосредственно в прикладной логике. Мы более подробно поговорим о явном управлении транзакциями в EJB далее в этой главе.

В случае применения декларативного управления транзакциями транзакционным поведением компонентов можно управлять с помощью дескриптора развертывания, в котором устанавливаются атрибуты транзакции для отдельных методов компонента. Это означает, что транзакционное поведение любого компонента может быть модифици-

ровано без изменения его прикладной логики. Кроме того, для компонента, разворачиваемого в одном приложении, может быть задано транзакционное поведение, отличающееся от поведения того же компонента, разворачиваемого в другом приложении. Декларативное управление транзакциями уменьшает сложность транзакций для разработчиков компонентов и приложений и облегчает создание мощных транзакционных приложений.

Зона транзакции

Зона транзакции (*transaction scope*) является важным понятием для понимания транзакций. В данном контексте зона транзакции обозначает компоненты – и сеансовые, и объектные, – участвующие в отдельной транзакции.

В методе `bookPassage()` компонента TravelAgent все имеющиеся компоненты являются частью одной зоны транзакции. Зона транзакции создается, когда клиент вызывает метод `bookPassage()` компонента TravelAgent. После этого зона транзакции *распространяется* (*is propagated*) на вновь созданные компоненты Reservation и ProcessPayment.

Как вы уже знаете, транзакция – это единица работы, состоящая из одной или нескольких задач. В транзакции все задачи, составляющие единицу работы, должны завершиться успешно, для того чтобы вся транзакция считалась успешной. Транзакция должна быть атомарной. Если какая-либо из задач потерпит неудачу, будет произведен откат (отмена) всех изменений, внесенных остальными задачами транзакции. В EJB задачи выражаются в терминах компонентных методов, а единица работы состоит из всех методов компонента, вызываемых в транзакции. Зона транзакции включает в себя все компоненты, участвующие в единице работы.

Достаточно просто проследить зону транзакции, следуя за потоком выполнения. Если обращение к методу `bookPassage()` начинает транзакцию, то, по логике, транзакция завершается после завершения данного метода. Зона транзакции `bookPassage()` должна включать в себя компоненты TravelAgent, Reservation и ProcessPayment – все компоненты, которых касается метод `bookPassage()`. Транзакция распространяется на компонент, когда вызывается метод этого компонента и включается в зону данной транзакции.

Транзакция может завершиться при возбуждении исключения во время выполнения метода `bookPassage()`. Исключение может быть возбуждено одним из компонентов или самим методом `bookPassage()`. Исключение может вызывать, а может и не вызывать откат, в зависимости от его типа. Мы рассмотрим более подробно исключения и транзакции далее.

Поток выполнения не является единственный фактором, определяющим, входит ли компонент в зону транзакции, атрибуты транзакции

компонента также играют свою роль. Принадлежность компонента к зоне какой-либо единицы работы определяется либо неявно с помощью атрибутов транзакции компонентов, либо явно с помощью транзакционного API Java (JTA).

Атрибуты транзакций

Разработчику приложения, как правило, не приходится явно управлять транзакциями при помощи сервера EJB. Серверы EJB могут управлять транзакциями неявно, опираясь на атрибуты транзакции, устанавливаемые для компонентов во время развертывания. Возможность указать с помощью программирования, основанного на атрибутах, как прикладные объекты должны функционировать в транзакциях, является общей характеристикой CTM и одной из наиболее важных особенностей компонентной модели EJB.

Во время развертывания компонента можно установить в дескрипторе развертывания его атрибут транзакции времени выполнения в одно из нескольких значений. В следующем списке показаны значения XML-атрибутов, предназначенные для определения этих атрибутов транзакций:

- `NotSupported`
- `Supports`
- `Required`
- `RequiresNew`
- `Mandatory`
- `Never`

Атрибуты транзакций упрощают построение транзакционных приложений, уменьшая риск, связанный с неправильным применением протоколов транзакций типа JTA (рассматриваемого ниже в этой главе). Использование атрибутов транзакций намного эффективнее и проще явного управления транзакциями.

Можно установить атрибут транзакции для всего компонента (в этом случае он будет применяться ко всем методам) или установить свои атрибуты транзакции для каждого отдельного метода. Первый метод намного проще и меньше подвержен ошибкам, но установка атрибутов на уровне методов дает большую гибкость. Фрагменты кода, приведенные в следующих разделах, показывают, как следует устанавливать в дескрипторе развертывания атрибут транзакции компонента по умолчанию.

Установка атрибута транзакции

В XML-дескрипторе развертывания элемент `<container-transaction>` определяет атрибуты транзакции для компонентов, описанных в этом дескрипторе развертывания:

```
<ejb-jar>
    ...
    <assembly-descriptor>
        ...
        <container-transaction>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
                <method-name> * </method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
            <method>
                <ejb-name>TravelAgentEJB</ejb-name>
                <method-name>listAvailableCabins</method-name>
            </method>
            <trans-attribute>Supports</trans-attribute>
        </container-transaction>
        ...
    </assembly-descriptor>
    ...
</ejb-jar>
```

Этот дескриптор развертывания определяет атрибуты транзакции для компонента TravelAgent. Каждый элемент `<container-transaction>` определяет метод и атрибут транзакции, который должен применяться к этому методу. Первый элемент `<container-transaction>` указывает, что по умолчанию все методы имеют атрибут транзакции `Required`; звездочка (*) является знаком подстановки, указывающим на все методы компонента TravelAgent. Второй элемент `<container-transaction>` переопределяет настройку по умолчанию и указывает для метода `listAvailableCabins()` атрибут транзакции `Supports`. Обратите внимание, что нам необходимо указать, на какой компонент мы ссылаемся в элементе `<ejb-name>`, т. к. XML-дескриптор развертывания может содержать несколько компонентов.

Возможные атрибуты транзакций

Здесь приведены описания атрибутов транзакций, перечисленных выше. В некоторых описаниях мы указываем, что транзакция клиента *приостановлена* (*suspended*). Это означает, что транзакция не распространяется на вызываемый метод компонента. Распространение транзакции временно приостановлено до возвращения из метода компонента. Для того чтобы вам было понятней, будем говорить о типах атрибутов, как будто они являются типами компонентов: например, мы будем говорить «компонент `Required`» в качестве сокращения для «компонент с атрибутом транзакции `Required`». Этими атрибутами являются:

NotSupported

Вызов метода компонента с этим атрибутом транзакции приостанавливает транзакцию до завершения метода. Это означает, что зона транзакции не распространяется ни на компонент NotSupported, ни на любой другой компонент, который он вызывает. После завершения метода компонента NotSupported свое выполнение возобновляет первоначальная транзакция.

На рис. 14.1 показано, что компонент NotSupported не распространяет транзакцию клиента при вызове одного из его методов.

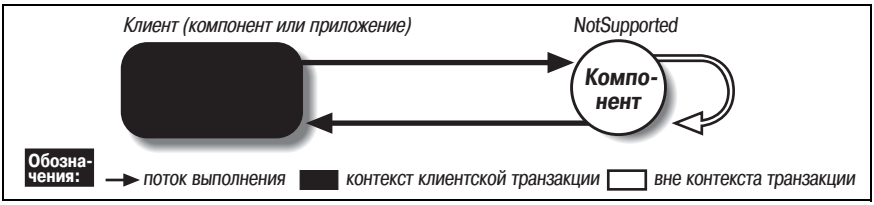

Рис. 14.1. Атрибут NotSupported

Supports

Этот атрибут означает, что метод компонента будет включен в зону транзакции, если он вызывается изнутри транзакции. Другими словами, если компонент или клиент, вызывающий компонент Supports, является частью зоны транзакции, компонент Supports и все компоненты, к которым он обращается, становятся частью первоначальной транзакции. Однако компонент Supports может не быть частью транзакции и может взаимодействовать с клиентами и другими компонентами, не входящими в зону транзакции.

На рис. 14.2(a) показан компонент Supports, вызываемый транзакционным клиентом и распространяющий транзакцию. На рис. 14.2(b) показан компонент Supports, вызываемый нетранзакционным клиентом.

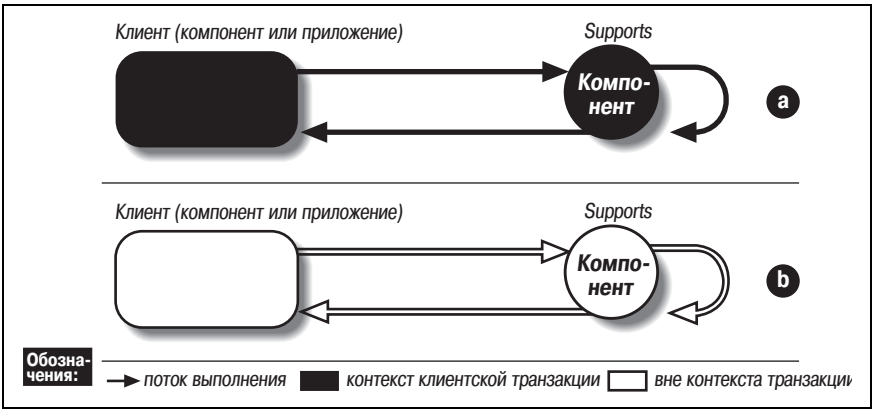

Рис. 14.2. Атрибут Supports

`Required`

Этот атрибут означает, что метод компонента должен вызываться изнутри транзакции. Если вызывающий клиент или компонент является частью транзакции, компонент `Required` автоматически включается в зону его транзакции. Однако если вызывающий клиент или компонент не связан ни с какой транзакцией, компонент `Required` начинает свою собственную новую транзакцию. Зона новой транзакции охватывает только компонент `Required` и все остальные компоненты, к которым он обращается. После того как вызванный метод компонента `Required` выполнен, зона новой транзакции закрывается.

На рис. 14.3(a) показан компонент `Required`, вызываемый транзакционным клиентом и распространяющий транзакцию.

На рис. 14.3(b) показан компонент `Required`, вызываемый нетранзакционным клиентом, который заставляет его начать свою собственную транзакцию.

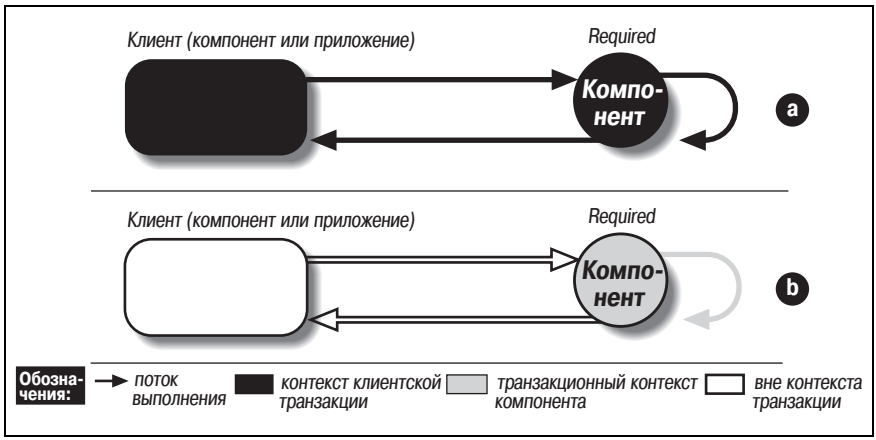

Рис. 14.3. Атрибут Required

`RequiresNew`

Этот атрибут означает, что всегда должна начинаться новая транзакция. Независимо от того, является ли вызывающий клиент или компонент частью транзакции, при своем вызове метод, имеющий атрибут `RequiresNew`, начинает новую транзакцию. Если вызывающий клиент уже связан с транзакцией, эта транзакция приостанавливается до возврата из метода компонента `RequiresNew`. Зона новой транзакции охватывает только компонент `RequiresNew` и все компоненты, к которым он обращается. После того как метод, вызванный для компонента `RequiresNew`, завершается, зона новой транзакции заканчивается, а первоначальная транзакция возобновляется.

На рис. 14.4(a) показан компонент RequiresNew, вызываемый транзакционным клиентом. Транзакция клиента приостанавливается на время выполнения компонента в его собственной транзакции.

На рис. 14.4(b) показан компонент RequiresNew, вызываемый нетранзакционным клиентом; компонент RequiresNew выполняется внутри своей собственной транзакции.

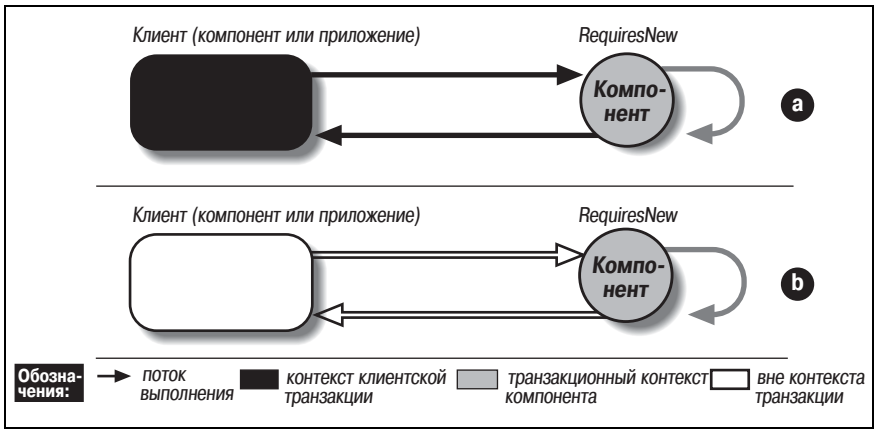

Рис. 14.4. Атрибут RequiresNew

Mandatory

Этот атрибут означает, что метод компонента должен всегда быть сделан частью зоны транзакции вызывающего клиента. Если вызывающий клиент или компонент не является частью транзакции, вызов приведет к возбуждению исключения javax.transaction.TransactionRequiredException для удаленного клиента или javax.ejb.TransactionRequiredLocalException – для локального клиента EJB 2.0.

На рис. 14.5(a) показан компонент Mandatory, вызываемый транзакционным клиентом и распространяющий транзакцию. На рис. 14.5(b) показан компонент Mandatory, вызываемый нетранзакционным клиентом. Метод генерирует TransactionRequiredException для удаленного клиента или TransactionRequredLocalException для локального клиента EJB 2.0, поскольку не существует никакой зоны транзакции.

Never

Этот атрибут означает, что метод компонента не должен вызываться изнутри транзакции. Если вызывающий клиент или компонент являются частью транзакции, компонент Never сгенерирует исключение RemoteException для удаленных клиентов или EJBException для локальных клиентов EJB 2.0. Однако если вызывающий клиент или компонент не связан с транзакцией, компонент Never выполнится как обычно – без транзакции.

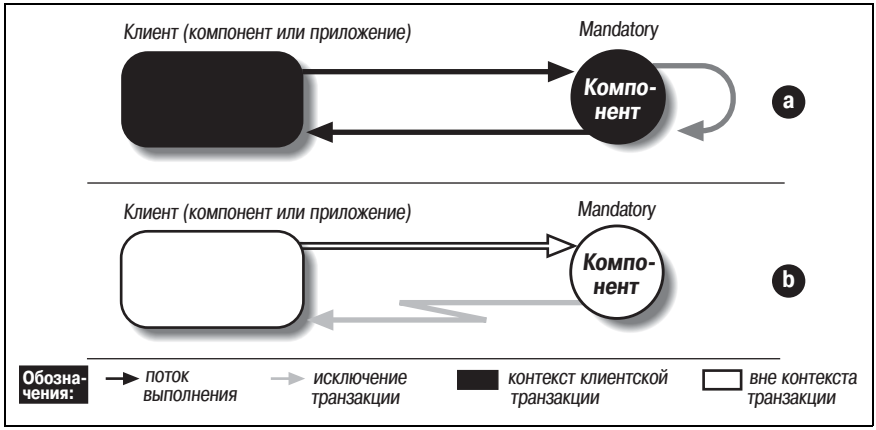

Рис. 14.5. Атрибут Mandatory

На рис. 14.6(a) показан компонент `Never`, вызываемый нетранзакционным клиентом. На рис. 14.6(b) показан компонент `Never`, вызываемый транзакционным клиентом, метод генерирует `RemoteException` для удаленных клиентов или `EJBException` для локальных клиентов EJB 2.0, т. к. метод никогда не может вызываться клиентом или компонентом, включенным в состав транзакции.

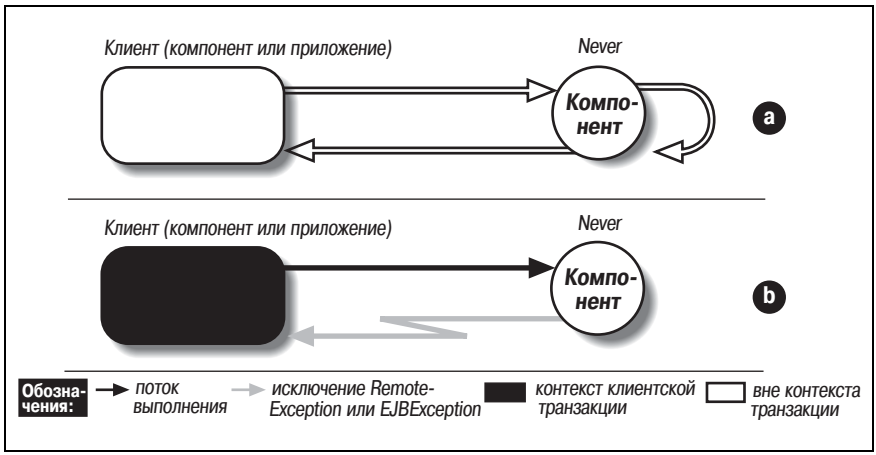

Рис. 14.6. Атрибут Never

EJB 2.0: Постоянство, управляемое контейнером, и атрибуты транзакции

Спецификация EJB 2.0 настоятельно рекомендует, чтобы объектные компоненты CMP 2.0 использовали только атрибуты транзакции `Required`, `RequiresNew` и `Mandatory`. Это ограничение гарантирует, что все обращения к базе данных происходят в контексте транзакции, что важ-

но при автоматическом управлении постоянством контейнером. Хотя спецификация требует, чтобы эти три атрибута транзакции поддерживались в CMP 2.0, поддержка для атрибутов `Never`, `Supports` и `NotSupported` не обязательна. Если производитель желает реализовать поддержку этих атрибутов (которые позволяют компонентам выполняться без транзакции), он может сделать это, хотя это и не рекомендуется. Сверьтесь с документацией, чтобы определить, поддерживаются ли необязательные атрибуты транзакции. Данная книга рекомендует, чтобы, имея дело с объектными компонентами, управляемыми контейнером, вы использовали только `Required`, `RequiresNew` и `Mandatory`.

EJB 2.0: Компоненты, управляемые сообщениями, и атрибуты транзакции

В компонентах, управляемых сообщениями, могут быть объявлены только атрибуты транзакции `NotSupported` и `Required`. Другие атрибуты в компонентах, управляемых сообщениями, не имеют смысла, т. к. они применяются к транзакциям, инициируемым клиентом. Атрибуты `Supports`, `RequiresNew`, `Mandatory` и `Never` все относятся к контексту транзакции клиента. Например, атрибут `Mandatory` требует, чтобы клиент уже имел запущенную транзакцию перед вызовом компонента. Это не имеет смысла для компонента, управляемого сообщениями, соединение которого с клиентом разорвано.

Атрибут транзакции `NotSupported` указывает, что сообщение будет обрабатываться без транзакции. Атрибут транзакции `Required` указывает, что сообщение будет обрабатываться внутри транзакции, инициированной контейнером.

Распространение транзакций

Покажем влияние атрибутов транзакции на методы компонента, для чего еще раз посмотрим на метод `bookPassage()` компонента TravelAgent, созданного в главе 12 (см. листинг выше в текущей главе).

Для того чтобы `bookPassage()` мог быть успешно выполнен в виде транзакции, и создание компонента Reservation, и прием платежа от пассажира также должны быть выполнены успешно. Это означает, что оба действия необходимо включить в одну транзакцию. Если какая-либо операция заканчивается неудачно, это вызывает сбой всей транзакции. Мы могли бы указать в качестве значения по умолчанию для всех вложенных компонентов атрибут транзакции `Required`, поскольку этот атрибут описывает требуемую нам политику, состоящую в том, что все компоненты должны выполниться внутри транзакции, и таким образом обеспечивает согласованность данных.

В качестве монитора транзакций сервер EJB наблюдает за каждым вызовом метода в пределах транзакции. Если происходит сбой в одной из операций обновления, все изменения всех компонентов будут отмене-

ны, или *прокручены назад* (*rolled back*). Откат похож на команду *отмены* (*undo*). Те, кому приходилось работать с реляционными базами данных, должны быть знакомы с понятием отката (rollback). Выполнив обновления, вы можете или зафиксировать их, или выполнить откат. Фиксация делает изменения, совершенные операцией обновления, постоянными. Откат прерывает обновление и оставляет базу данных в ее первоначальном состоянии. Создание транзакционных компонентов представляет собой такой же вид управления откатом/фиксацией. Например, если компонент Reservation не может быть создан, платеж, принятый компонентом ProcessPayment, отменяется. Транзакции выполняют изменения по принципу «все или ничего». Это гарантирует, что единица работы, такая как метод `bookPassage()`, будет выполнена так, как и предполагалось, и будет предотвращена запись несогласованных данных в базу данных.

В случаях, когда контейнер управляет транзакциями неявно, решения о применении фиксации или отката принимаются автоматически. Если имеет место явное управление транзакциями внутри компонента или они управляются клиентом, ответственность за фиксации и откаты ложится на разработчика компонента или приложения. Явное задание границ транзакций подробно рассматривается ниже в этой главе.

Предположим, что компонент TravelAgent создан и используется на стороне клиента следующим образом:

```
TravelAgent agent = agentHome.create(customer);
agent.setCabinID(cabin_id);
agent.setCruiseID(cruise_id);
try {
    agent.bookPassage(card,price);
} catch(Exception e) {
    System.out.println("Transaction failed!");
}
```

Более того, предположим, что методу `bookPassage()` присвоен атрибут транзакции `RequiresNew`. В этом случае клиент, вызывающий метод `bookPassage()`, сам не является частью транзакции. Когда вызывается метод `bookPassage()` компонента TravelAgent, создается новая транзакция, как и задано атрибутом `RequiresNew`. Это означает, что компонент TravelAgent регистрирует себя в менеджере транзакций сервера EJB, который будет автоматически управлять данной транзакцией. Менеджер транзакций координирует транзакции, распространяя зоны транзакции от одного компонента до другого, чтобы обеспечить включение всех компонентов, имеющих отношение к транзакции, в ее единицу работы. Таким способом менеджер транзакций может контролировать обновления, сделанные каждым компонентом, и, основываясь на результате этих обновлений, принимать решение о фиксации в базе данных всех изменений, внесенных компонентами, или о выполнении отката. Если методом `bookPassage()` возбуждается *системное исключе-*

ние, транзакция автоматически отменяется. Мы поговорим более подробно об исключениях ниже в этой главе.

Когда из метода `bookPassage()` вызывается метод `byCredit()`, компонент ProcessPayment регистрируется в менеджере транзакций внутри контекста транзакции, которая была создана для компонента TravelAgent. Транзакционный контекст распространяется на компонент ProcessPayment. Создаваемый вновь компонент Reservation также регистрируется в менеджере транзакций внутри той же транзакции. После того как зарегистрированы все компоненты и выполнено их обновление, менеджер транзакций убеждается в том, что их обновления будут работать. Если все обновления будут работать, менеджер транзакций разрешает зафиксировать эти изменения. Если один из компонентов сообщает об ошибке или сбое, все изменения, сделанные компонентами ProcessPayment или Reservation, отменяются менеджером транзакций. На рис. 14.7 показано распространение и управление транзакционным контекстом компонента TravelAgent.

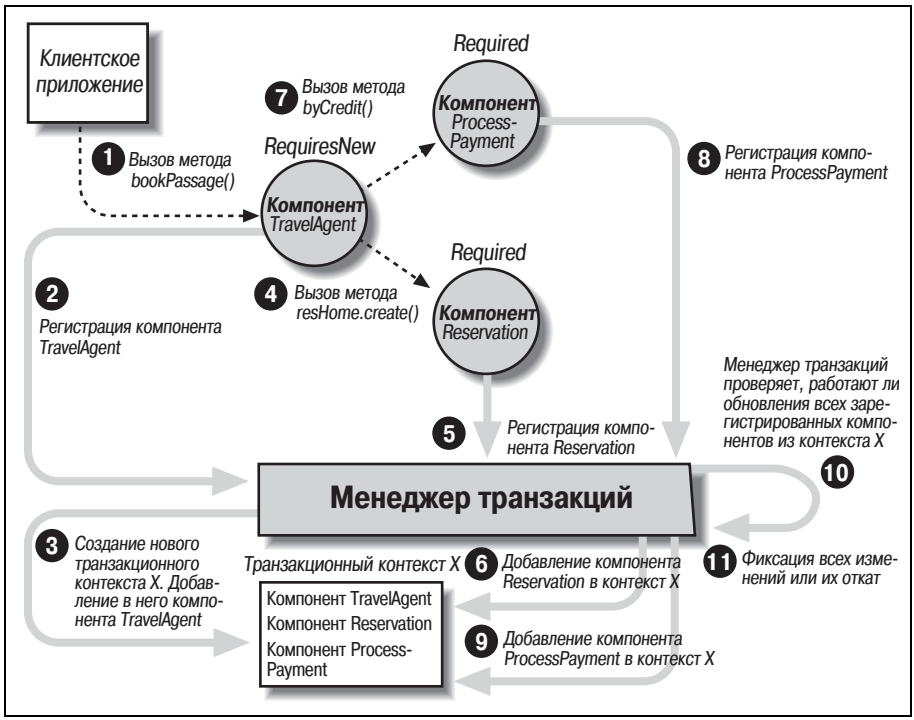

Рис. 14.7. Управление транзакционным контекстом компонента TravelAgent

Кроме управления транзакциями в своем собственном окружении, сервер EJB может взаимодействовать с другими системами транзакций. Если бы, например, компонент ProcessPayment в действительности находился на другом сервере EJB, а не там, где компонент Tra-

velAgent, эти два сервера EJB могли бы совместно управлять транзакцией как одной единицей работы. Этот механизм получил название *распределенной транзакции* (*distributed transaction*).[1]

Распределенная транзакция является гораздо более сложной и требует так называемой *двухфазовой фиксации* (*two-phase commit, 2-PC или TPC*). Двухфазовая фиксация – это механизм, позволяющий управлять транзакциями между разными серверами и ресурсами (например, базами данных и провайдерами JMS). Описание деталей двухфазовой фиксации выходит за рамки данной книги, но система, поддерживающая их, не будет требовать никаких дополнительных операций со стороны разработчика компонента или приложения. Если распределенные транзакции поддерживаются, протокол для распространения транзакций, как было рассмотрено выше, будет также поддерживаться. Другими словами, как разработчик приложения или компонента вы, скорее всего, не увидите разницы между локальными и распределенными транзакциями.

EJB 2.0: Основанные на коллекции отношения и транзакции

В постоянстве, управляемом контейнером EJB 2.0, к отношениям, основанным на коллекции, можно обращаться только изнутри одиночной транзакции. Другими словами, будет ошибкой получать объект `Collection` из основанного на коллекции поля отношения в одной транзакции, а затем использовать его в другой.

Например, если компонент обращается к основанному на коллекции полю отношения другого компонента через его локальный интерфейс, объект `Collection`, возвращенный из метода доступа, может использоваться только внутри этой же транзакции:

```
public class HypotheticalBean implements javax.ejb.EntityBean {

    public void methodX(CustomerLocal customer) {

        Collection reservations = customer.getReservations();
        Iterator iterator = reservations.iterator;
        while(iterator.hasNext()){
            ...
        ...
    }
...
}
```

Если метод `getReservations()` компонента Customer был объявлен с атрибутом транзакции `RequiresNew`, попытка вызвать какой-либо метод

[1] Не все серверы EJB поддерживают распределенные транзакции.

объекта Collection, в том числе и метод iterator(), вызовет исключение java.lang.IllegalStateException. Это исключение возбуждается, т. к. объект Collection был создан внутри транзакции getReservations(), а не транзакции methodX(). Контекст транзакции methodX() отличается от контекста транзакции метода getReservations().

Объект Collection объектного компонента может использоваться другим совмещенным компонентом, только если он получен и функционирует в том же контексте транзакции. Если метод getReservations() компонента Customer распространяет контекст транзакции метода methodX(), то работа с объектом Collection не вызывает каких-либо проблем. Это может быть сделано с помощью изменения метода getReservations() путем объявления атрибута транзакции как Required или Mandatory.

Изоляция и блокировка базы данных

Изоляция транзакций (буква «I» – Isolation в аббревиатуре ACID) представляет собою чрезвычайно важный элемент любой системы транзакций. В данном разделе рассматриваются условия изоляции, блокировка базы данных и уровни изоляции транзакций. Эти понятия важны при развертывании любой системы транзакций.

Грязное, повторяющееся и фантомное чтение

Изоляция транзакций определяется в терминах условий изоляции называемых *грязным чтением* (*dirty read*), *повторяющимся чтением* (*repeatable read*) и *фантомным чтением* (*phantom read*). Эти условия описывают, что может случаться, когда две или более транзакции обращаются к одним и тем же данным.[1]

Для того чтобы проиллюстрировать эти условия, представим себе два отдельных клиентских приложения, использующие свои собственные экземпляры компонента TravelAgent для доступа к одним и тем же данным, а именно, к записи, относящейся к каюте, с первичным ключом 99. Эти примеры строятся вокруг таблицы RESERVATION, к которой обращаются и метод bookPassage() (через компонент Reservation), и метод listAvailableCabins() (через JDBC). Хорошо было бы вернуться и вспомнить, как в этих методах осуществляется доступ к таблице RESERVATION. Это поможет вам понять, как две транзакции, выполняемые двумя разными клиентами, могут влиять друг на друга. Допустим, что оба метода имеют атрибут транзакции Required.

[1] Условия изоляции подробно описаны в спецификации ANSI SQL-92, номер документа: ANSI X3.135-1992 (R1998).

Грязное чтение

Грязное чтение происходит тогда, когда одна транзакция читает незакрепленные изменения, сделанные другой транзакцией. Если выполняется откат второй транзакции, данные, считанные первой транзакцией, становятся недействительными, поскольку в результате отката отменяются все изменения. Первая транзакция не будет знать, что данные, которые она прочитала, стали недействительными. Ниже приведен сценарий, показывающий, как может происходить грязное чтение (рис. 14.8).

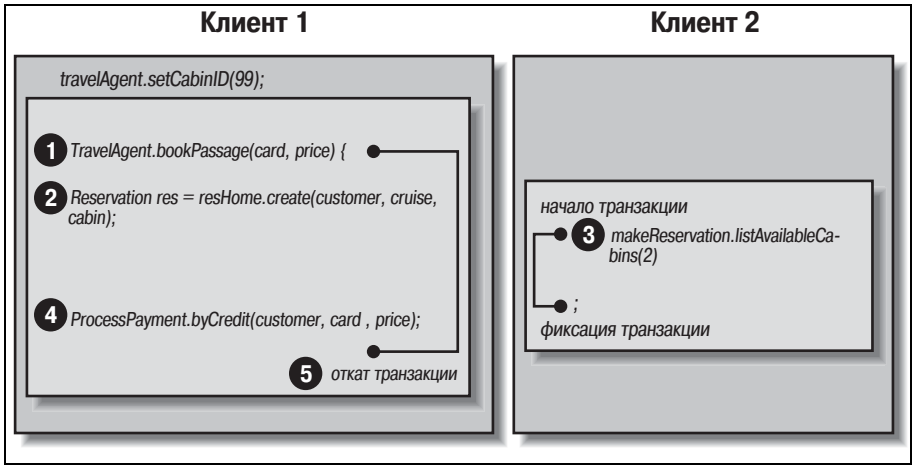

Рис. 14.8. Грязное чтение

1. Время 10:00:00: Клиент 1 выполняет метод `TravelAgent.bookPassage()`. Вместе с компонентами Customer и Cruise Клиент 1 ранее выбрал каюту 99 для включения в заказ.
2. Время 10:00:01: Компонент TravelAgent Клиента 1 создает внутри метода `bookPassage()` компонент Reservation. Метод `create()` компонента Reservation вставляет запись в таблицу `RESERVATION`, которая резервирует каюту 99.
3. Время 10:00:02: Клиент 2 выполняет метод `TravelAgent.listAvailableCabins()`. Каюта 99 уже зарезервирована Клиентом 1, поэтому ее нет в списке свободных кают, возвращаемом этим методом.
4. Время 10:00:03: Компонент TravelAgent Клиента 1 внутри метода `bookPassage()` выполняет метод `ProcessPayment.byCredit()`. Метод `byCredit()` генерирует исключение из-за истечения срока годности переданной ему кредитной карточки.
5. Время 10:00:04: Исключение, возбужденное компонентом ProcessPayment, приводит к откату всей транзакции `bookPassage()`. В результате запись, вставленная в таблицу `RESERVATION` при создании компонента Reservation, не делается устойчивой (т. е. удаляется). Каюта 99 теперь свободна.

Клиент 2 теперь имеет дело с недействительным списком свободных кают, т. к. каюта 99 свободна, но не включена в список. Это упущение может стать достаточно серьезным, если каюта 99 является последней свободной каютой, потому что Клиент 2 дал бы неверное сообщение о том, что круиз полностью укомплектован. Пассажир, возможно, попытается заказать круиз у конкурентов.

Повторяющееся чтение

Повторяющееся чтение происходит тогда, когда читаемые данные гарантированно выглядят точно так же при повторном чтении внутри той же транзакции. Повторяющееся чтение обеспечивается одним из двух способов: либо при чтении данных происходит их блокировка от изменений, либо осуществляется чтение снимка данных, который не отражает изменения. Если данные блокированы, они не могут быть изменены никакой другой транзакцией до окончания текущей транзакции. Если данные представляют собой снимок, другие транзакции могут изменять данные, но эти изменения не будут замечены транзакцией при повторном чтении. Ниже приведен пример повторяющегося чтения (рис. 14.9).

1. Время 10:00:00: Клиент 1 начинает явную транзакцию `javax.transaction.UserTransaction`.
2. Время 10:00:01: Клиент 1 выполняет метод `TravelAgent.listAvailableCabins(2)`, запрашивая список свободных кают, имеющих две кровати. Каюта 99 содержится в списке свободных кают.
3. Время 10:00:02: Клиент 2 работает с интерфейсом, который управляет компонентами Cabin. Клиент 2 пытается изменить количество спальных мест в каюте 99 с 2 на 3.
4. Время 10:00:03: Клиент 1 заново выполняет метод `TravelAgent.listAvailableCabins(2)`. Каюта 99 все еще в списке свободных кают.

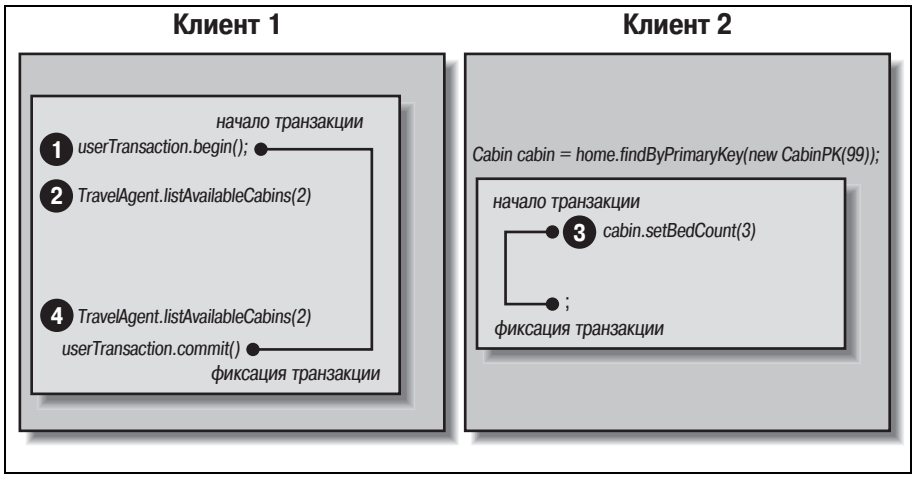

Рис. 14.9. Повторяющееся чтение

Этот пример несколько необычен из-за того, что он использует `javax.transaction.UserTransaction`. Этот класс рассматривается более подробно далее в этой главе. По существу, он позволяет клиентскому приложению управлять зоной транзакции явно. В нашем случае Клиент 1 размещает границы транзакции вокруг обоих вызовов метода `listAvailableCabins()` так, чтобы они были частью одной транзакции. Если бы Клиент 1 не сделал это, два метода `listAvailableCabins()` выполнились бы как отдельные транзакции, и наше условие повторяющегося чтения не наступило бы.

Хотя Клиент 2 пытался изменить количество коек в каюте 99 до 3, каюта 99 все же появляется в результате запроса Клиентом 1 метода `listAvailableCabins()`, в котором запрашивалось 2 койки. Это происходит из-за того, что либо Клиенту 2 не дали внести изменения (из-за блокировки), либо Клиент 2 смог внести изменение, но Клиент 1 работает со снимком данных, на который это изменение не повлияло.

Неповторяющееся чтение (*nonrepeatable read*) имеет место тогда, когда данные, возвращаемые последующим чтением внутри одной транзакции, могут представлять разные результаты. Другими словами, последующее чтение может видеть изменения, сделанные другими транзакциями.

Фантомное чтение

Фантомные чтения происходят тогда, когда новые записи, добавляемые к базе данных, обнаруживаются транзакциями, начавшимися до этой вставки. В запросы будут включены записи, добавленные другими транзакциями после того, как их транзакция началась. Ниже приведен сценарий, содержащий фантомное чтение (рис. 14.10).

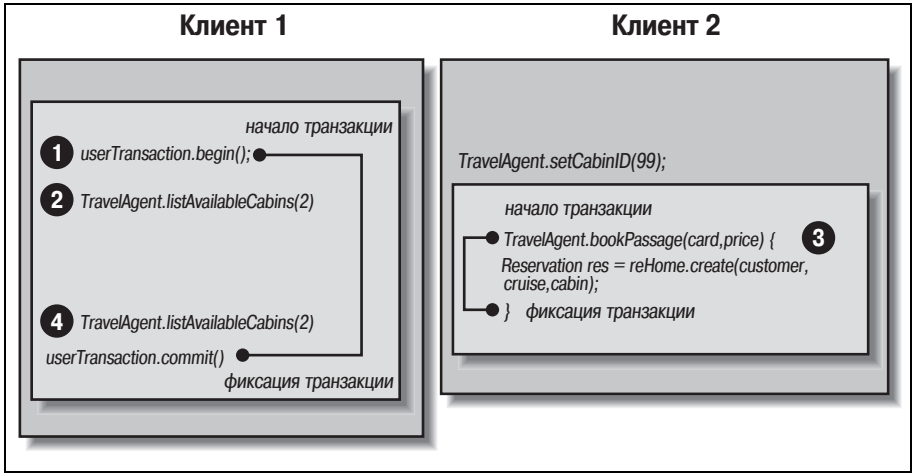

Рис. 14.10. Фантомное чтение

1. Время 10:00:00: Клиент 1 начинает явную транзакцию `javax.transaction.UserTransaction`.
2. Время 10:00:01: Клиент 1 выполняет метод `TravelAgent.listAvailableCabins(2)`, запрашивая список свободных кают, имеющих два спальных места. Каюта 99 находится в списке свободных кают.
3. Время 10:00:02: Клиент 2 выполняет `bookPassage()` и создает компонент Reservation. Резервирование вставляет новую запись в таблицу `RESERVATION`, резервируя каюту 99.
4. Время 10:00:03: Клиент 1 повторно выполняет метод `TravelAgent.listAvailableCabins(2)`. Каюты 99 больше нет в списке свободных кают.

Клиент 1 устанавливает границы транзакции вокруг обоих вызовов метода `listAvailableCabins()` так, чтобы они были частью одной транзакции. В этом случае заказ билетов был сделан между запросами `listAvailableCabins()` внутри одной транзакции. Следовательно, запись, вставленная в таблицу `RESERVATION`, не существовала во время вызова первого метода `listAvailableCabins()`, но она уже существовала и была видна во время второго вызова метода `listAvailableCabins()`. Эта вставленная запись называется *фантомной* (*phantom record*).

Блокировки баз данных

Базы данных, особенно реляционные базы данных, обычно используют несколько разных методов блокировки. Наиболее распространенными являются *блокировки чтения* (*read locks*), *блокировки записи* (*write locks*) и *монопольные блокировки записи* (*exclusive write locks*). (Я позволил себе добавить снимки данных (snapshots), хотя они и не являются формальным термином.) Эти механизмы блокировки определяют, каким способом транзакции будут одновременно обращаться к данным. Механизмы блокировки влияют на только что описанные условия чтения. Эти типы блокировок представляют собой достаточно простые понятия, которые непосредственно не связаны со спецификацией EJB. Производители баз данных реализуют эти блокировки поразному, поэтому необходимо понять, как ваша база данных реализует эти механизмы блокировки, для того чтобы лучше представлять, как будут работать уровни изоляции, описанные в этом разделе.

Существуют четыре типа блокировок:

Блокировки чтения

Блокировки чтения не разрешают другим транзакциям изменять данные, читаемые транзакцией, до ее завершения, таким образом предотвращая неповторяющееся чтение. Другие транзакции могут читать эти данные, но не могут изменять их. Текущей транзакции также запрещается внесение изменений. Будет ли блокировка чтения блокировать чтение записей, блока записей или целой таблицы, зависит от базы данных.

Блокировки записи

Блокировки записи используются при обновлениях. Блокировка записи запрещает другим транзакциям изменять данные до окончания текущей транзакции, но разрешает выполнять грязное чтение другим транзакциям и самой текущей транзакции. Другими словами, транзакция может читать свои собственные незакрепленные изменения.

Монопольные блокировки записи

Монопольные блокировки записи используются при обновлениях. Монопольная блокировка записи запрещает другим транзакциям читать и модифицировать данные до завершения текущей транзакции. Такая блокировка предотвращает грязное чтение другими транзакциями. Другим транзакциям не разрешается читать данные, пока они исключительно блокированы. Некоторые базы данных не разрешают транзакциям читать свои собственные данные, пока они блокированы монопольно.

Снимки данных

Некоторые базы данных обходятся без блокировок, предоставляя каждой транзакции собственный снимок данных. Снимок данных – это зафиксированное представление данных, взятое в начале транзакции. Снимки могут предотвращать грязное чтение, неповторяющееся и фантомное чтение. Они могут привести к проблемам из-за того, что данные в них не являются данными реального времени, они представляют данные в момент получения снимка.

Уровни изоляции транзакций

Изоляция транзакции задается в терминах условий изоляции (грязное чтение, повторяющееся чтение и фантомное чтение). Уровни изоляции обычно используются в системах баз данных для описания того, какая блокировка будет применяться к данным внутри транзакции.[1] При рассмотрении уровней изоляции обычно оперируют следующими терминами:

Незакрепленное чтение

Транзакция может читать незакрепленные данные (данные, измененные другой транзакцией, которая все еще выполняется).

Могут иметь место грязное чтение, неповторяющееся чтение и фантомное чтение. Методы компонента с данным уровнем изоляции могут читать незакрепленные изменения.

[1] Условия изоляции подробно описаны в спецификации ANSI SQL-92, номер документа: ANSI X3.135-1992(R1998).

Закрепленное чтение

Транзакция не может читать незакрепленные данные. Данные, модифицируемые другой транзакцией, не могут быть прочитаны.

Грязное чтение запрещается. Могут иметь место неповторяющееся и фантомное чтение. Методы компонента с данным уровнем изоляции не могут читать незакрепленные данные.

Повторяющееся чтение

Транзакция не может модифицировать данные, читаемые другой транзакцией.

Грязное и неповторяющееся чтения запрещены. Может иметь место фантомное чтение. Методы компонента с данным уровнем изоляции имеют те же самые ограничения, что и при закрепленном чтении, и могут выполнять только повторяющееся чтение.

Сериализуемый

Транзакция имеет монопольные права на чтение и обновление данных. Другие транзакции не могут ни читать, ни изменять эти данные.

Грязное, неповторяющееся и фантомное чтения запрещены. Данный уровень изоляции является наиболее ограниченным.

Те же уровни изоляции определены и для JDBC, а именно: они соответствуют статическим неизменяемым переменным в классе `java.sql.Connection`. Поведение, моделируемое уровнями изоляции в данном классе, не отличается от того, которое было описано здесь.

Точное поведение этих уровней изоляции в значительной степени зависит от механизма блокировки, используемого базой данных или ресурсом. Функционирование уровней изоляции в значительной степени определяется тем, как их поддерживает ваша база данных.

В EJB управляющий развертыванием устанавливает уровни изоляции транзакций способом, специфичным для конкретного производителя, если транзакциями управляет контейнер. Если компонент сам управляет своими транзакциями, уровень изоляции транзакций устанавливается разработчиком данного компонента. До этого момента мы рассматривали только транзакции, управляемые контейнером. Мы рассмотрим транзакции, управляемые компонентом, далее в этой главе.

Выбор между производительностью и согласованностью

Вообще говоря, по мере того как уровни изоляции становятся более ограничивающими, производительность системы уменьшается, поскольку более ограничивающие уровни изоляции не дают транзакциям обращаться к одним и тем же данным. Если уровень изоляции оказывает сильное ограничивающее действие, как, например, сериализуемый,

то все транзакции, даже просто считывающие, должны стоять в очереди на выполнение. В результате система может стать очень медленной. Системы EJB обрабатывают большое количество параллельных транзакций и должны быть очень быстрым, поэтому в них необходимо избегать применения сериализуемого уровня изоляции там, где это не обязательно, т. к. он является крайне медленным.

Однако вместе с тем, уровни изоляции обеспечивают согласованность данных. Более ограничивающие уровни изоляции помогают гарантировать, что для выполнения обновлений не будут использованы недействительные данные. Как говорится в старой пословице, «из двух зол выбирают меньшее». Сериализуемый уровень изоляции гарантирует, что к данным никогда не будут обращаться несколько транзакций одновременно, таким образом обеспечивается согласованность данных.

Выбор подходящего уровня изоляции требует определенного знания используемой базы данных и особенностей управления блокировкой в ней. Необходимо также найти баланс между производительностью системы и согласованностью данных. Эта задача не имеет универсального решения из-за того, что разные прикладные программы используют данные по-разному.

Хотя в системе «Титан» присутствуют только три судна, объектные компоненты, которые представляют их, входят в большинство транзакций «Титана». Это означает, что в одно и то же время к этим компонентам Ship будут обращаться несколько, возможно, сотни транзакций. Доступ к компонентам Ship должен быть быстрым, иначе они станут «узким местом», поэтому мы не должны использовать слишком ограничивающий уровень изоляции. В то же время, данные, относящиеся к судну, должны быть согласованными. В противном случае в сотнях транзакций будут участвовать недействительные данные. Следовательно, при внесении изменений в информацию о кораблях необходимо задать строгий уровень изоляции. Чтобы совместить эти противоречивые требования, мы можем применять к разным методам различные уровни изоляции.

Большинство транзакций используют для получения информации методы get компонента Ship. Это операция *только для чтения* (*read-only*), поэтому уровень изоляции для полученных методов может быть очень низким, вроде незакрепленного чтения. Методы set компонента Ship почти никогда не применяются. Название судна, возможно, не будет меняться годами. Однако данные, изменяемые методами set, должны быть изолированы, чтобы предотвратить грязное чтение со стороны других транзакций, поэтому для методов set судна мы зададим наиболее ограничивающий уровень изоляции – сериализуемый. Выбирая для разных прикладных методов разные уровни изоляции, мы можем балансировать между согласованностью данных и производительностью.

Управление уровнями изоляции

Разные серверы EJB при установке уровней изоляции предлагают разные степени точности. Некоторые серверы перекладывают эту ответственность на базу данных. Большинство серверов EJB управляют уровнем изоляции через API доступа к ресурсам (например, JDBC и JMS) и могут разрешать разным ресурсам иметь разные уровни изоляции, но обычно требуют, чтобы для доступа к одному ресурсу в пределах отдельной транзакции был установлен согласованный уровень изоляции. Уровень управления, предлагаемый вашим сервером, можно уточнить в документации по нему.

Однако транзакции, управляемые компонентом, в сеансовых компонентах (с состоянием и без состояния) и компонентах, управляемых сообщениями (EJB 2.0), позволяют разработчику компонента задавать уровень изоляции транзакций с помощью API ресурса, предоставляющего постоянное хранилище. API для JDBC, например, предоставляет механизм задания уровня изоляции для подключения к базе данных. Следующий код показывает, как это можно сделать. Транзакции, управляемые компонентом, рассмотрены более подробно далее в этой главе.

```
...
DataSource source = (javax.sql.DataSource)
    jndiCntxt.lookup("java:comp/env/jdbc/titanDB");

Connection con = source.getConnection();
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
...
```

Вы можете установить разный уровень изоляции для разных ресурсов в пределах одной транзакции, но все компоненты, использующие один и тот же ресурс внутри транзакции, должны использовать одинаковый уровень изоляции.

Нетранзакционные компоненты

Компоненты, находящиеся вне зоны транзакции, обычно предоставляют какой-либо сервис, не требующий состояния, не управляющий данными непосредственно в хранилище данных. Хотя эти типы компонентов могут быть необходимы в качестве утилит внутри транзакции, они не обязаны удовлетворять строгим требованиям ACID-транзакций.

Рассмотрим нетранзакционный сеансовый компонент без состояния Quote, предоставляющий реальные биржевые котировки. Этот компонент может отвечать на запросы транзакционного компонента, связанного с транзакцией оптовой закупки. Успех или неудача оптовой закупки, как транзакции, не будут влиять на состояние или действия

компонента Quote, поэтому он не должен быть частью транзакции. Для компонентов, входящих в состав транзакции, установлено свойство «изолированные», а это означает, что их сервисами *нельзя* пользоваться одновременно во время действия транзакции. Создание транзакционного компонента может быть достаточно дорогим действием во время выполнения. Объявление компонента нетранзакционным (`NotSupported`) выводит его за пределы зоны транзакции, что может улучшить производительность и доступность его сервиса.

Явное управление транзакциями

Хотя в этом разделе рассматривается JTA, строго рекомендуется не пытаться управлять транзакциями явно. Через атрибуты транзакции Enterprise JavaBeans предоставляет всесторонний и простой механизм для разграничивания транзакций на уровне методов и автоматического распространения транзакций. Только разработчики, глубоко понимающие транзакционные системы, могут попробовать использовать JTA в EJB.

В EJB неявное управление транзакциями представлено на уровне методов компонентов, чтобы мы могли определять транзакции, которые ограничиваются областью выполняемого метода. Оно является одним из основных преимуществ EJB над более «сырыми» реализациями распределенных объектов: оно уменьшает сложность системы и, следовательно, количество ошибок программиста. Кроме того, декларативное установление границ транзакций, как это используется в EJB, отделяет транзакционные операции от прикладной логики. Изменение поведения транзакции не требует изменений прикладной логики. В редких случаях, однако, может потребоваться контролировать транзакции явно. Чтобы сделать это, необходимо всестороннее понимание транзакций.

Явное управление транзакциями достаточно сложно и обычно выполняется с помощью службы объектных транзакций (Object Transaction Service, OTS) OMG или Java-реализации OTS – службы транзакций Java (Java Transaction Service, JTS). OTS и JTS предоставляют API, позволяющий разработчикам обращаться непосредственно к менеджерам и ресурсам транзакций (например, базам данных и провайдерам JMS). Хотя JTS-реализация OTS представляет собой мощный и законченный инструмент, это – не самый простой API для работы. Он требует явного и специального контроля над областями выполнения транзакций.

Enterprise JavaBeans поддерживает для работы с транзакциями гораздо более простой прикладной интерфейс – Java Transaction API (JTA). Этот API реализуется пакетом `javax.transaction`. В действительности

JTA состоит из двух компонентов: интерфейса транзакционного клиента высокого уровня и интерфейса нижнего уровня X/Open XA. Нас интересует интерфейс клиента высокого уровня, так как только он доступен для компонентов и рекомендован как транзакционный интерфейс для клиентских приложений. Интерфейс нижнего уровня XA используется сервером и контейнером EJB для автоматической координации транзакций с ресурсами, такими как базы данных.

Как разработчик приложений и компонентов, вы вероятно, предпочтете явное управление транзакциями, уделяя внимание одному очень простому интерфейсу: `javax.transaction.UserTransaction`. `UserTransaction` предоставляет механизм взаимодействия с менеджером транзакций, позволяющий разработчику приложения управлять зоной транзакции явно. Ниже приведен пример явного установления границ транзакций в компоненте или клиентском приложении:

```
Object ref = getInitialContext().lookup("TravelAgentHomeRemote");
TravelAgentHome home = (TravelAgentHome)
    PortableRemoteObject.narrow(ref, TravelAgentHome.class);

TravelAgent tr1 = home.create(customer);
tr1.setCruiseID(cruiseID);
tr1.setCabinID(cabin_1);
TravelAgent tr2 = home.create(customer);
tr2.setCruiseID(cruiseID);
tr2.setCabinID(cabin_2);

// получаем UserTransaction
javax.transaction.UserTransaction tran = ...;

tran.begin();
tr1.bookPassage(visaCard,price);
tr2.bookPassage(visaCard,price);
tran.commit();
```

Клиентскому приложению требуется зарезервировать две каюты для одного пассажира. В нашем случае пассажир заказывает каюту для себя и своих детей. Клиент не хочет заказывать никакую каюту, если он не может получить обе, поэтому клиентское приложение разработано так, чтобы поместить оба заказа в одну транзакцию. Это достигается явным указанием границ транзакции с помощью объекта `javax.transaction.UserTransaction`. Каждый метод компонента, вызванный текущим потоком между вызовами методов `UserTransaction.begin()` и `UserTransaction.commit()`, включается в одну и ту же зону транзакции в соответствии с атрибутами транзакции вызываемых методов компонента.

Надуманность примера очевидна, но смысл становится ясен. Транзакциями можно управлять непосредственно, а не зависеть от зоны методов, разграничивающих их. Преимущество явного установления границ транзакций в том, что оно дает клиенту контроль над границами транзакции. Клиентом в данном случае могло бы быть клиентское

приложение или другой компонент.[1] В обоих случаях используется один и тот же объект javax.transaction.UserTransaction, но он извлекается из разных источников в зависимости от того, кому он требуется – клиенту или компоненту.

Java 2 Enterprise Edition (J2EE) определяет, как клиентское приложение может с помощью JNDI получить объект UserTransaction. Далее показано, как клиент получает объект UserTransaction, если контейнер EJB является частью системы J2EE (J2EE и ее связь с EJB более подробно рассмотрены в главе 17):

```
...
Context jndiCntx = new InitialContext();
UserTransaction tran = (UserTransaction)
    jndiCntx.lookup("java:comp/UserTransaction");
utx.begin();
...
utx.commit();
...
```

Компоненты также могут управлять транзакциями явно. Только сеансовые компоненты и компоненты, управляемые сообщениями, со значением <transaction-type>, установленным в Bean, могут управлять своими собственными транзакциями. Компоненты, управляющие своими собственными транзакциями, часто называют компонентами, управляющими транзакциями самостоятельно (bean-managed transaction, BMT). Объектные компоненты никогда не могут быть BMT-компонентами. BMT-компоненты не задают атрибуты транзакции для своих методов. Ниже показано, как сеансовый компонент объявляет, что он будет управлять транзакциями явно:

```
<ejb-jar>
    <enterprise-beans>
    ...
        <session>
        ...
        <transaction-type>Bean</transaction-type>
        ...
```

Чтобы управлять своей собственной транзакцией, компонент должен получить объект UserTransaction. Компонент получает ссылку на UserTransaction из EJBContext, как показано здесь:

```
public class HypotheticalBean extends SessionBean {
    SessionContext ejbContext;

    public void someMethod() {
```

[1] Интерфейс UserTransaction могут использовать только компоненты, объявленные как самостоятельно управляющие своими транзакциями (BMT-компоненты).

```
        try {
            UserTransaction ut = ejbContext.getUserTransaction();
            ut.begin();

            // выполняем какую-нибудь работу

            ut.commit();
    } catch(IllegalStateException ise) {...}
        catch(SystemException se) {...}
        catch(TransactionRolledbackException tre) {...}
        catch(HeuristicRollbackException hre) {...}
        catch(HeuristicMixedException hme) {...}
```

Компонент может также обратиться к `UserTransaction` из JNDI ENC, как показано в следующем примере. Оба метода подходят для этой цели. Компонент выполняет поиск, используя контекст «`java:comp/env/UserTransaction`»:

```
InitialContext jndiCntx = new InitialContext();
UserTransaction tran = (UserTransaction)
    jndiCntx.lookup("java:comp/env/UserTransaction");
```

Распространение транзакций в транзакциях, управляемых компонентом

В случае с сеансовыми компонентами без состояния транзакции, которыми управляют с помощью `UserTransaction`, должны начинаться и заканчиваться внутри одного и того же метода. Другими словами, транзакции `UserTransaction` не могут быть начаты в одном методе, а закончены в другом. Это имеет смысл, поскольку экземпляры сеансового компонента без состояния используются одновременно несколькими клиентами, поэтому пока один экземпляр без состояния может обслуживать первый запрос клиента, следующий запрос этого же клиента может обслуживать совершенно другой экземпляр. Однако в случае с сеансовыми компонентами с состоянием транзакция может начаться в одном методе, а быть зафиксированной – в другом, т. к. сеансовый компонент с состоянием используется только одним клиентом. Это позволяет сеансовому компоненту с состоянием связать себя с транзакцией, используемой несколькими разными методами, вызываемыми клиентами. Представьте себе, к примеру, компонент TravelAgent в виде BMT-компонента. В следующем коде транзакция начинается в методе `setCruiseID()`, а заканчивается в методе `bookPassage()`. Это позволяет методам компонента TravelAgent быть связанными с одной и той же транзакцией.

EJB 2.0: TravelAgentBean

Определение класса `TravelAgentBean` в EJB 2.0 выглядит так:

```
import com.titan.reservation.*;

import java.sql.*;
```

```
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;

public class TravelAgentBean implements javax.ejb.SessionBean {
    ...
    public void setCruiseID(Integer cruiseID)
        throws javax.ejb.FinderException {
        try {
            ejbContext.getUserTransaction().begin();
            CruiseHomeLocal home = (CruiseHomeLocal)
                jndiContext.lookup("java:comp/env/ejb/CruiseHome");

            cruise = home.findByPrimaryKey(cruiseID);
        } catch(RemoteException re) {
            throw new EJBException(re);
        }

    }
    public TicketDO bookPassage(CreditCardDO card, double price)
        throws IncompleteConversationalState {

        try {
            if (ejbContext.getUserTransaction().getStatus() !=
                javax.transaction.Status.STATUS_ACTIVE) {

                throw new EJBException("Transaction is not active");
            }
        } catch(javax.transaction.SystemException se) {
             throw new EJBException(se);
        }

        if (customer == null || cruise == null || cabin == null)
        {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeLocal resHome = (ReservationHomeLocal)
               jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price);

            Object ref =
                jndiContext.lookup("java:comp/env/ejb/
                   ProcessPaymentHomeRemote");

            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                PortableRemoteObject.narrow(ref,
                   ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
```

```
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

            ejbContext.getUserTransaction().commit();

            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    ...
}
```

EJB 1.1: TravelAgentBean

В EJB 1.1 определение класса `TravelAgentBean` выглядит так:

```
public class TravelAgentBean implements javax.ejb.SessionBean {
    ...
    public void setCruiseID(Integer cruiseID)
        throws javax.ejb.FinderException {
        try {
            ejbContext.getUserTransaction().begin();
            CruiseHomeRemote home = (CruiseHomeRemote)
                getHome("CruiseHome", CruiseHomeRemote. class);
            cruise = home.findByPrimaryKey(cruiseID);
        } catch(RemoteException re) {
            throw new EJBException(re);
        }

    }
    public TicketDO bookPassage(CreditCardDO card, double price)
        throws IncompleteConversationalState {

        try {
            if (ejbContext.getUserTransaction().getStatus() !=
                javax.transaction.Status.STATUS_ACTIVE) {

                throw new EJBException("Transaction is not active");
            }
        } catch(javax.transaction.SystemException se) {
            throw new EJBException(se);
        }

        if (customer == null || cruise == null || cabin == null) {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeRemote resHome = (ReservationHomeRemote)
              getHome("ReservationHomeRemote", ReservationHomeRemote.class);
            ReservationRemote reservation =
                resHome.create(customer, cruise, cabin, price);
```

```
            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                getHome("ProcessPaymentHomeRemote",
                        ProcessPaymentHomeRemote.class);
            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

            ejbContext.getUserTransaction().commit();

            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    ...
}
```

Повторные вызовы метода `EJBContext.getUserTransaction()` возвращают ссылку на один и тот же объект `UserTransaction`. От контейнера требуется до завершения транзакции сохранять связь между транзакцией и экземпляром компонента с состоянием между множественными клиентскими вызовами.

В методе `bookPassage()` мы можем проверить состояние транзакции, для того чтобы гарантировать, что она все еще активна. Если транзакция больше не активна, мы генерируем исключение. Применение метода `getStatus()` более подробно рассмотрено позже в этой главе.

Когда управляемый компонентом метод транзакции вызывается клиентом, который уже является частью транзакции, транзакция клиента приостанавливается до возвращения из метода. Приостановка происходит независимо от того, начал ли явно BMT-компонент внутри метода свою собственную транзакцию, или транзакция была начата в предыдущем вызове метода. Транзакция клиента всегда приостанавливается до возвращения из BMT-метода.

Настоятельно не рекомендуем распределять управление транзакциями по нескольким методам, т. к. это может привести к некорректно управляемым транзакциям и длительной блокировке ресурсов.

EJB 2.0: Компоненты, управляемые сообщениями, и транзакции, управляемые компонентами

Компоненты, управляемые сообщениями, также имеют возможность управлять своими собственными транзакциями. В случае с MDB зона транзакции должна начинаться и заканчиваться внутри метода `onMessage()`: управляемая компонентом транзакция не может быть распределена по нескольким вызовам `onMessage()`.

Не составляет труда преобразовать компонент ReservationProcessor в BMT-компонент, просто изменив его значение <transaction-type> на Bean:

```
<ejb-jar>
    <enterprise-beans>
    ...
        <message-driven>
        ...
        <transaction-type>Bean</transaction-type>
        ...
```

В данном случае класс ReservationProcessorBean может быть изменен так, чтобы использовать javax.transaction.UserTransaction для обозначения начала и конца транзакции в onMessage():

```
public class ReservationProcessorBean implements
javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void onMessage(Message message) {
        try {

            ejbContext.getUserTransaction().begin();

            MapMessage reservationMsg = (MapMessage)message;

            Integer customerPk =
                (Integer)reservationMsg.getObject("CustomerID");
            Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
            Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");
            double price = reservationMsg.getDouble("Price");

            //get the credit card
            Date expirationDate =
                new Date(reservationMsg.getLong("CreditCardExpDate"));
            String cardNumber = reservationMsg.getString("CreditCardNum");
            String cardType = reservationMsg.getString("CreditCardType");

            CreditCardDO card =
                new CreditCardDO(cardNumber,expirationDate,cardType);
            CustomerRemote customer = getCustomer(customerPk);
            CruiseLocal cruise = getCruise(cruisePk);
            CabinLocal cabin = getCabin(cabinPk);

            ReservationHomeLocal resHome = (ReservationHomeLocal)
               jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");
            ReservationLocal reservation =
                resHome.create(customer,cruise,cabin,price,new Date());

            Object ref =
```

```
            jndiContext.lookup("java:comp/env/ejb/
                ProcessPaymentHomeRemote");
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();

        process.byCredit(customer,card,price);
        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
        deliverTicket(reservationMsg,ticket);

        ejbContext.getUserTransaction.commit();

    } catch(Exception e) {
        throw new EJBException(e);
    }
}
...
```

Важно понять, что в BMT сообщение, получаемое MDB, не является частью транзакции. Когда MDB использует управляемые контейнером транзакции, сообщение, которое он обрабатывает, является частью транзакции. Поэтому, если выполняется откат транзакции, откатывается также и получение сообщения, что вынуждает провайдер JMS послать сообщение повторно. Но в случае с транзакциями, управляемыми компонентом, сообщение не является частью транзакции, поэтому если BMT-транзакция отменяется, провайдер JMS не будет знать о неудачном завершении транзакции. Однако еще не все потеряно, т. к. провайдер JMS все еще может рассчитывать на подтверждение сообщения, чтобы определить, было ли оно доставлено успешно.

Контейнер EJB подтверждает получение сообщение, если метод `onMessage()` завершается успешно. Однако если методом `onMessage()` возбуждается исключение `RuntimeException`, контейнер не подтвердит получение сообщения, и провайдер JMS не узнает о возникшей проблеме и, вероятно, попытается послать сообщение повторно. Если повторная отправка сообщения важна, в случае сбоя транзакции в BMT лучше убедиться, что метод `onMessage()` генерирует `EJBException`, т. к. контейнер не будет подтверждать сообщение, полученное от провайдера JMS.

Производители используют собственные (декларативные) механизмы для указания количества попыток повторной отправки сообщений для компонентов BMT/NotSupported MDB, которые «отказываются» подтверждать получение. Провайдер JMS может предоставлять зону «мертвых сообщений», в которую будут помещаться такие сообщения, если они не могут быть успешно обработаны в соответствии со счетчиком повторений. Зона мертвых сообщений может наблюдаться администратором, и доставленные сообщения могут быть обнаружены и обработаны вручную.

Сообщение не является частью транзакции, но все остальное, заключенное между методами `UserTransaction.begin()` и `UserTransaction.commit()`, представляет собою часть одной транзакции. Это включает в себя создание нового компонента Reservation и обработку кредитной карточки с помощью компонента ProcessPayment. Если происходит сбой транзакции, будет выполнен откат этих действий. Транзакция также включает использование JMS API в методе `deliverTicket()` для посылки сообщения о билете. В случае сбоя транзакции сообщение о билете не будет отослано.

Эвристические решения

Транзакциями обычно управляет *менеджер транзакций* (*transaction manager*), роль которого часто выполняет сервер EJB, управляющий ACID-характеристиками нескольких компонентов, баз данных и серверов. Для управления транзакциями этот менеджер транзакций использует двухфазовую фиксацию 2-PC – протокол для управления транзакциями, который выполняет закрепление обновлений в две стадии. Этот протокол достаточно сложен, но в основном он требует, чтобы серверы и базы данных взаимодействовали друг с другом через посредника, менеджера транзакций, для того чтобы гарантировать, что все данные делаются устойчивыми одновременно. Некоторые серверы EJB поддерживают 2-PC, а другие – нет, и ценность этого механизма транзакций вызывает некоторые споры. Важно запомнить, что менеджер транзакций управляет транзакцией. Опираясь на результаты опроса ресурсов (баз данных, провайдеров JMS и других), он решает, нужно ли закреплять все сделанные обновления или выполнить откат. *Эвристическое решение* имеет место тогда, когда один из ресурсов принимает одностороннее решение о закреплении или откате без получения разрешения от менеджера транзакций. После того как эвристическое решение принято, атомарность транзакции теряется, и могут иметь место ошибки в целостности данных.

Интерфейс `UserTransaction`, рассматриваемый в следующем разделе, генерирует несколько разных исключений, связанных с эвристическими решениями. Они включены в последующее обсуждение.

UserTransaction

`UserTransaction` – это интерфейс Java, определенный в следующем коде. От серверов EJB не требуется поддержка остальной части JTA, и для их служб транзакций не требуется применение JTS. `UserTransaction` определен следующим образом:

```
public interface javax.transaction.UserTransaction {

    public abstract void begin() throws IllegalStateException,
        SystemException;
```

```
    public abstract void commit() throws IllegalStateException,
        SystemException, TransactionRolledbackException,
        HeuristicRollbackException, HeuristicMixedException;
    public abstract int getStatus();
    public abstract void rollback() throws IllegalStateException,
        SecurityException, SystemException;
    public abstract void setRollbackOnly() throws IllegalStateException,
        SystemException;
    public abstract void setTransactionTimeout(int seconds) throws
        SystemException;

}
```

Далее описано действие методов, определенных в этом интерфейсе:

`begin()`

Вызов метода `begin()` создает новую транзакцию. Поток, выполняющий метод `begin()`, немедленно связывается с новой транзакцией, которая затем распространяется на все компоненты, поддерживающие существующие транзакции. Метод `begin()` может генерировать одно из двух проверяемых исключений. Исключение `IllegalStateException` возбуждается, когда `begin()` вызывается потоком, который уже связан с какой-либо транзакцией. Перед началом новой транзакции необходимо завершить все транзакции, связанные с этим потоком. Исключение `SystemException` возбуждается, если менеджер транзакции (т. е. EJB-сервер) сталкивается с неожиданной аварийной ситуацией.

`commit()`

Метод `commit()` завершает транзакцию, связанную с текущим потоком. Когда `commit()` выполняется, текущий поток больше не связан с транзакцией. Данный метод может генерировать несколько перехватываемых исключений. `IllegalStateException` возбуждается, если текущий поток не связан с транзакцией. `SystemException` возбуждается, если менеджер транзакции (EJB-сервер) сталкивается с неожиданной аварийной ситуацией. `TransactionRolledbackException` возбуждается, когда вместо закрепления происходит откат всей транзакции. Это может произойти, если один из ресурсов не смог выполнить обновление, или если был вызван метод `UserTransaction.rollBackOnly()`. `HeuristicRollbackException` указывает, что одним или несколькими ресурсами были приняты эвристические решения об откате транзакции. `HeuristicMixedException` указывает, что ресурсами были приняты эвристические решения и об откате, и о закреплении транзакции, т. е. частью ресурсов было принято решение об откате, в то время как другие решили выполнить закрепление.

`rollback()`

Метод `rollback()` вызывается для отката транзакции и отмены всех обновлений. Метод `rollback()` может возбуждать одно из трех различных проверяемых исключений. `SecurityException` возбуждается, если потоку, использующему объект `UserTransaction`, не разрешено выполнять откат транзакции. `IllegalStateException` возбуждается, если текущий поток не связан с транзакцией. `SystemException` возбуждается, если менеджер транзакций (EJB-сервер) сталкивается с неожиданной аварийной ситуацией.

`setRollbackOnly()`

Метод `setRollbackOnly()` вызывается для того, чтобы отметить транзакцию как подлежащую откату. Это означает, что независимо от результата обновлений, сделанных внутри транзакции, по завершении транзакции должен быть выполнен ее откат. Этот метод может быть вызван любым компонентом `TX_BEAN_MANAGED`, являющимся частью транзакции или клиентского приложения. Метод `setRollBackOnly()` может генерировать одно из двух перехватываемых исключений. Исключение `IllegalStateException` возбуждается, если текущий поток не связан с транзакцией. `SystemException` возбуждается, если менеджер транзакции (EJB-сервер) сталкивается с неожиданной аварийной ситуацией.

`setTransactionTimeout(int seconds)`

Метод `setTransactionTimeout(int seconds)` устанавливает продолжительность жизни транзакции, т. е. определяет, как долго она будет жить до истечения ее времени. Транзакция должна завершиться до достижения «тайм-аута». Если этот метод не был вызван, менеджер транзакции (EJB-сервер) устанавливает «тайм-аут» автоматически. Если этот метод вызывается со значением 0 секунд, менеджер транзакций будет использовать «тайм-аут» по умолчанию. Этот метод должен вызываться после метода `begin()`. Исключение `SystemException` возбуждается, если менеджер транзакции (EJB-сервер) сталкивается с неожиданной аварийной ситуацией.

`getStatus()`

Метод `getStatus()` возвращает целое число, которое необходимо сравнить с константами, определенными в интерфейсе `javax.transaction.Status`. Этот метод должен использоваться только опытными программистами для определения статуса транзакции, связанной с объектом `UserTransaction`. `SystemException` возбуждается, если менеджер транзакции (EJB-сервер) сталкивается с неожиданной аварийной ситуацией.

Состояние

`Status` является довольно простым интерфейсом, не содержащим никаких методов, а только константы. Его единственное предназначение

состоит в предоставлении набора констант, описывающих текущее состояние транзакционного объекта, в данном случае – UserTransaction:

```
interface javax.transaction.Status
{
    public final static int STATUS_ACTIVE;
    public final static int STATUS_COMMITTED;
    public final static int STATUS_COMMITTING;
    public final static int STATUS_MARKED_ROLLBACK;
    public final static int STATUS_NO_TRANSACTION;
    public final static int STATUS_PREPARED;
    public final static int STATUS_PREPARING;
    public final static int STATUS_ROLLEDBACK;
    public final static int STATUS_ROLLING_BACK;
    public final static int STATUS_UNKNOWN;
}
```

Значение, возвращаемое методом getStatus(), сообщает клиенту, использующему UserTransaction, статус транзакции. Далее приводятся значения этих констант:

STATUS_ACTIVE

С объектом UserTransaction связана активная транзакция. Это состояние возвращается после того, как транзакция была начата, и до того как менеджер транзакций начал двухфазовую фиксацию. (Транзакции, которые были приостановлены, все еще считаются активными.)

STATUS_COMMITTED

Транзакция связана с объектом UserTransaction. Транзакция была закреплена. Вероятно, были приняты эвристические решения; иначе транзакция была бы прервана и возвращена константа STATUS_NO_TRANSACTION.

STATUS_COMMITTING

Транзакция связана с объектом UserTransaction. Транзакция находится в процессе закрепления. Объект UserTransaction возвращает этот статус, если менеджер транзакций решил выполнить закрепление, но еще не закончил процесс.

STATUS_MARKED_ROLLBACK

Транзакция связана с объектом UserTransaction. Транзакция была отмечена как подлежащая откату, возможно, в результате операции UserTransaction.setRollbackOnly(), вызванной где-то в приложении.

STATUS_NO_TRANSACTION

В настоящее время с объектом UserTransaction не связана ни одна транзакция. Это происходит после завершения транзакции, или если

не было создано ни одной транзакции. Это значение возвращается вместо возбуждения исключения `llegalStateException`.

`STATUS_PREPARED`

Транзакция связана с объектом `UserTransaction`. Транзакция была подготовлена, а это означает, что первая стадия процесса двухфазовой фиксации завершена.

`STATUS_PREPARING`

Транзакция связана с объектом `UserTransaction`. Транзакция находится в процессе подготовки, а это означает, что менеджер транзакций находится в процессе выполнения первой стадии двухфазовой фиксации.

`STATUS_ROLLEDBACK`

Транзакция связана с объектом `UserTransaction`. Результат транзакции был идентифицирован как откат. Вероятно, были приняты эвристические решения; иначе транзакция была бы прервана и возвращена константа `STATUS_NO_TRANSACTION`.

`STATUS_ROLLING_BACK`

Транзакция связана с объектом `UserTransaction`. Транзакция находится в процессе отката.

`STATUS_UNKNOWN`

Транзакция связана с объектом `UserTransaction`. Ее текущее состояние не может быть определено. Это условие является переходным, и последующие обращения в конечном счете возвратят другой статус.

Методы отката EJBContext

Только BMT-компоненты имеют доступ к `UserTransaction` из `EJBContext` и JNDI ENC. Компоненты с транзакциями, управляемыми контейнером (container-managed transaction, CTM), не могут использовать `UserTransaction`. Взаимодействие CMT-компонентов с текущей транзакцией обеспечивается методами `setRollbackOnly()` и `getRollbackOnly()` объекта `EJBContext`.

Метод `setRollbackOnly()` дает компоненту право наложить вето на транзакцию. Это право может быть использовано в случае, если компонент обнаружит условие, которое может привести к тому, что при завершении транзакции будут закреплены несогласованные данные. Если компонент вызывает метод `setRollbackOnly()`, текущая транзакция помечается как подлежащая откату и не может быть закреплена никаким другим участником транзакции, включая контейнер.

Если текущая транзакция была отмечена как подлежащая откату, метод `getRollbackOnly()` возвращает значение `true`. Это позволяет избежать

выполнения работы, которая в любом случае не будет закреплена. Если, например, исключение было возбуждено и перехвачено внутри метода компонента, то посредством метода `getRollbackOnly()` можно определить, привело ли исключение к откату текущей транзакции. Если это имело место, нет никакого смысла продолжать обработку. Если этого не произошло, компонент имеет возможность исправить ошибку и повторить задачу, завершившуюся неуспехом. Только опытные разработчики компонентов могут прибегать к повторному выполнению задач в пределах одной транзакции. Если исключение не вызывало отката (`getRollbackOnly()` возвращает `false`), то альтернатива состоит в том, что можно форсировать откат с помощью метода `setRollbackOnly()`.

BMT-компоненты *не могут* использовать методы `setRollbackOnly()` и `getRollbackOnly()` объекта `EJBContext`. Для проверки и форсирования отката BMT-компоненты должны обращаться к методам `getStatus()` и `rollback()` объекта `UserTransaction`, соответственно.

Исключения и транзакции

Исключения оказывают существенное влияние на результат транзакций и должны быть подробно рассмотрены для того, чтобы разработчики компонентов поняли отношения между ними.

Сравнение прикладных и системных исключений

Прикладное исключение – это любое исключение, которое не расширяет `java.lang.RuntimeException` или `java.rmi.RemoteException`. Системные исключения – это `java.lang.RuntimeException` и его подтипы, включая `EJBException`.

Прикладное исключение никогда не должно расширять ни `RuntimeException`, ни `RemoteException`, ни один из их подтипов.

Если метод компонента возбуждает системное исключение, выполняется *автоматический* откат транзакции. Транзакции *не* откатываются автоматически, если возбуждается прикладное исключение. Тот, кто запомнит эти два правила, будет готов к работе с исключениями и транзакциями в EJB.

Метод `bookPassage()` хорошо иллюстрирует прикладное исключение и особенности его применения. Код метода `bookPassage()` показан в следующих разделах.

EJB 2.0: Метод bookPassage()

Для EJB 2.0:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref =
           jndiContext.lookup("java:comp/env/ejb/ProcessPaymentHomeRemote");

        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
           PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

EJB 1.1: Метод bookPassage()

Для EJB 1.1:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome = (ReservationHomeRemote)
            getHome("ReservationHomeRemote", ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            getHome("ProcessPaymentHomeRemote",
            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);
```

```
            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
```

Системные исключения

Системные исключения – это `RuntimeException` и его подклассы. Исключение `EJBException` представляет собой подкласс `RuntimeException` и поэтому считается системным.

Системные исключения при возбуждении в методе компонента всегда вызывают откат транзакции. Все исключения `RuntimeException` (`EJBException`, `NullPointerException`, `IndexOutOfBoundsException` и т. д.), возбуждаемые в пределах метода `bookPassage()`, автоматически обрабатываются контейнером и вызывают откат транзакции. В Java типы `RuntimeException` не требуют ни объявления в разделе `throws` прототипа метода, ни обработки с помощью блоков `try/catch`. Они возбуждаются методом автоматически.

Типы `RuntimeException`, возбуждаемые внутри компонентов, всегда вызывают откат текущей транзакции. Если метод, в котором происходит исключение, запустил транзакцию, транзакция отменяется. Если транзакция была начата клиентом, вызвавшим метод, транзакция клиента помечается как подлежащая откату и не может быть закреплена.

Системные исключения обрабатываются автоматически контейнером, который всегда:

- Выполняет откат транзакции
- Регистрирует исключение, чтобы уведомить о нем системного администратора
- Удаляет экземпляр компонента

Исключения `RuntimeException`, возбуждаемые в методах обратного вызова (`ejbLoad()`, `ejbActivate()` и т. д.), обрабатываются так же, как и исключения, возбуждаемые прикладными методами.

EJB требует регистрации системных исключений, но не указывает ни способ, которым должны регистрироваться исключения, ни формат журнала. Реализация конкретного механизма регистрации исключений и сообщения о них системному администратору оставлена на усмотрение производителя.

При возникновении системного исключения экземпляр компонента уничтожается, а это означает, что удаляются все указывающие на него ссылки, и он удаляется сборщиком мусора. Контейнер предполагает, что экземпляр компонента может содержать испорченные переменные

или может быть нестабильным и поэтому небезопасным для использования в каком-то еще аспекте.

Влияние уничтожения экземпляра компонента зависит от типа компонента. В случае сеансовых компонентов без состояния и объектных компонентов клиент даже не заметит, что экземпляр был удален. Эти типы экземпляров не назначаются определенному клиенту. Они перемещаются из пула экземпляров и обратно, так что вновь поступивший запрос может обслужить любой экземпляр. Однако в случае с сеансовыми компонентами с состоянием воздействие на клиента гораздо серьезнее. Сеансовые компоненты с состоянием выделяются для одного клиента и поддерживают состояние диалога с ним. Удаление экземпляра компонента с состоянием уничтожает состояние диалога экземпляра и делает клиентскую ссылку на данный компонент недействительной. Когда уничтожаются сеансовые экземпляры с состоянием, последующие вызовы методов компонентов клиентом приводят к возбуждению исключения `NoSuchObjectException`, являющегося подклассом `RemoteException`.[1]

В случае с компонентами, управляемыми сообщениями, системное исключение, возбуждаемое в методе `onMessage()` или одном из методов обратного вызова (`ejbCreate()` или `ejbRemove()`), приводит к уничтожению экземпляра компонента. Если MDB является BMT-компонентом, обрабатываемое сообщение может быть послано повторно (или не послано), в зависимости от того, когда контейнер EJB подтвердит доставку. В случае транзакций, управляемых контейнером, контейнер выполнит откат транзакций, поэтому сообщение не будет подтверждено и может быть послано повторно.

В сеансовых и объектных компонентах, когда возникает системное исключение и экземпляр удаляется, на удаленных клиентах всегда возбуждается исключение `RemoteException`, т. е. на клиентах, использующих удаленные интерфейсы компонентов. Если клиент начинает транзакцию, которая затем распространяется на компонент, системное исключение (возбужденное методом компонента) будет перехвачено контейнером и возбуждено повторно в виде `javax.transaction.TransactionRolledbackException`. `TransactionRolledbackException` представляет собой подтип `RemoteException`. Оно более явно показывает клиенту, что произошел откат транзакции.

В сеансовых компонентах EJB 2.0 и объектных компонентах при возникновении системного исключения и удалении экземпляра во всех локальных клиентах компонента (т. е. клиентах, использующих локальные интерфейсы компонента) всегда возбуждается `EJBException`.

[1] Хотя возникновение исключения `RemoteException` всегда приводит к удалению экземпляра, воздействие на удаленную ссылку может сильно различаться для разных производителей.

Если клиент начал транзакцию и она затем была распространена на компонент, системное исключение (возбужденное методом компонента) будет перехвачено контейнером и повторно возбуждено в виде javax.ejb.TransactionRolledbackLocalException. TransactionRolledbackLocalException представляет собой подтип EJBException. Оно более явно показывает клиенту, что произошел откат транзакции. В других случаях, независимо от того, управляется ли компонент контейнером или самостоятельно, исключение RuntimeException, возбужденное изнутри метода компонента, будет перехвачено контейнером и повторно возбуждено в виде EJBException.

Как правило, EJBException возбуждается, когда подсистема генерирует исключение, например SQLException от JDBC или JMSException от JMS. Однако в некоторых случаях разработчик компонента может попробовать самостоятельно обработать исключение и повторить операцию, а не генерировать EJBException. Это следует делать, только если вы хорошо понимаете исключения, возбуждаемые подсистемой, и их воздействие на транзакцию. Есть эмпирическое правило: повторно генерируйте исключения подсистемы в виде EJBExceptions и разрешите контейнеру EJB выполнять откат транзакций и удалять экземпляры компонентов.

Методы обратного вызова, определенные в интерфейсах javax.ejb.EntityBean и javax.ejb.SessionBean, объявляют в своих секциях throws исключение java.rmi.RemoteException. Оно осталось от EJB 1.0 и было отменено в EJB 1.1. Не следует генерировать RemoteExceptions в методах обратного вызова или каких-либо других методах класса компонента.

Прикладные исключения

Прикладное исключение обычно возбуждается в ответ на ошибку в прикладной логике, а не на системную ошибку. Прикладные исключения всегда посылаются непосредственно клиенту, без повторной упаковки в типы RemoteException или EJBException (EJB 2.0). Обычно они не вызывают откат транзакции; как правило, клиент имеет возможность восстановиться после возбуждения прикладного исключения. Например метод bookPassage() генерирует прикладное исключение с именем IncompleteConversationalState. Это прикладное исключение, поскольку оно не расширяет ни RuntimeException, ни RemoteException. Исключение IncompleteConversationalState возбуждается в случае, если один из параметров, переданных методу bookPassage(), содержит null. (Прикладные ошибки часто используются для сообщения об ошибках при проверке действительности, как в данном случае.) В этом случае исключение возбуждается перед началом выполнения задачи, и ясно, что оно не является результатом отказа подсистемы (например, JDBC, JMS, Java RMI, JNDI).

Из-за того что это – прикладное исключение, возбуждение исключения IncompleteConversationalState не вызовет откат транзакции. Исключение возбуждается до того, как будет выполнена какая-либо работа, обходя ненужное выполнение метода bookPassage() и давая возможность клиенту (компоненту или приложению, вызвавшему метод bookPassage()) восстановиться и, возможно, повторить вызов метода с правильными параметрами.

Прикладные методы, определенные в удаленных и локальных интерфейсах, могут генерировать любой вид прикладных исключений. Эти прикладные исключения должны быть объявлены в сигнатурах методов удаленных и локальных интерфейсов и в соответствующих методах классов компонентов.

Конструирующие, поисковые и удаляющие методы компонента также могут генерировать некоторые исключения, определенные в пакете javax.ejb: CreateException, DuplicateKeyException, FinderException, ObjectNotFoundException и RemoveException. Эти исключения считаются прикладными исключениями: они передаются клиенту как есть, без повторной упаковки в виде RemoteExceptions. Кроме того, данные исключения не обязательно приводят к откату транзакции, предоставляя клиенту возможность повторить операцию. Эти исключения могут возбуждаться непосредственно компонентами. В случае с постоянством, управляемым контейнером контейнер также может генерировать любое из этих исключений во время обработки конструирующих, поисковых и удаляющих методов компонентов (ejbCreate(), ejbFind() и ejbRemove()). Контейнер может, например, возбудить исключение CreateException, если он сталкивается с ошибочным параметром при попытке вставить запись для компонента, управляемого контейнером. Всегда можно возбудить стандартное прикладное исключение в нужном методе независимо от вида управления постоянством.

Далее приведено детальное описание пяти стандартных прикладных исключений и ситуаций, в которых они возбуждаются.

CreateException

CreateException возбуждается методом create() удаленного интерфейса. Это исключение может быть возбуждено контейнером, если контейнер управляет постоянством, или явно разработчиком компонента в методах ejbCreate() или ejbPostCreate(). Оно указывает, что во время создания компонента возникла прикладная ошибка (неверные параметры и т. п.). Если это исключение генерируется контейнером, то он может произвести откат транзакции. Для получения определенных результатов должны использоваться явные методы транзакций. Разработчики компонентов должны перед возбуждением этого исключения выполнять откат транзакции, только если он связан с целостностью данных.

`DuplicateKeyException`

`DuplicateKeyException` представляет собой подтип `CreateException`. Он возбуждается методом `create()` удаленного интерфейса. Это исключение может быть возбуждено контейнером, если контейнер управляет постоянством, или явно разработчиком компонента в методе `ejbCreate()`. Оно указывает, что в базе данных уже существует компонент с таким же первичным ключом. Перед возбуждением этого исключения компонент или контейнер, как правило, не выполняет откат транзакции.

`FinderException`

`FinderException` возбуждается поисковыми методами внутреннего интерфейса. Это исключение может быть возбуждено контейнером, если контейнер управляет постоянством, или явно разработчиком компонента в методах `ejbFind()`. Оно указывает, что при попытке поиска компонентов контейнером возникла прикладная ошибка (неправильные параметры и т. п.). Не применяйте этот метод для указания на то, что объекты не были найдены. Многообъектные поисковые методы возвращают пустую коллекцию, если не был найден ни один объект. Однообъектные поисковые методы для указания на то, что не был найден ни один объект, генерируют `ObjectNotFoundException`. Перед возбуждением этого исключения компонент или контейнер, как правило, не выполняет откат транзакции.

`ObjectNotFoundException`

`ObjectNotFoundException` возбуждается однообъектным поисковым методом для указания на то, что контейнер не смог найти требуемый объект. Это исключение может быть возбуждено контейнером (если контейнер управляет постоянством) или явно разработчиком компонента в методах `ejbFind()`. Оно не должно возбуждаться для указания на ошибку в прикладной логике (неправильные параметры и т. п.). Для указания на ошибки прикладной логики в однообъектных поисковых методах используйте исключение `FinderException`. `ObjectNotFoundException` возбуждается однообъектными поисковыми методами только для указания на то, что требуемый объект не был найден. Поисковые методы, возвращающие несколько объектов, должны возвращать пустую коллекцию, если ничего не было найдено. Перед возбуждением этого исключения компонент или контейнер, как правило, не выполняет откат транзакции.

`RemoveException`

Исключение `RemoveException` возбуждается в методах `remove()` удаленных и внутренних интерфейсов. Это исключение может быть возбуждено контейнером, если контейнер управляет постоянством, или явно разработчиком компонента в методе `ejbRemove()`. Оно указывает на возникновение прикладной ошибки при удалении компо-

нента. Перед возбуждением этого исключения контейнер, как правило, не выполняет откат транзакции. Для получения предсказуемого результата должны использоваться явные методы транзакции. Разработчики компонентов должны выполнять откат транзакции перед возбуждением этого исключения, только если оно связано с целостностью данных.

В табл. 14.1 представлен краткий обзор взаимоотношений между разными типами исключений и транзакций в сеансовых и объектных компонентах.

Таблица 14.1. Обзор исключений в сеансовых и объектных компонентах

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Представление клиента
Транзакция, инициируемая клиентом. Транзакция начинается клиентом (приложением или компонентом) и распространяется на методы компонента.	Тип транзакции = Контейнерная Атрибуты транзакции = `Required` | `Mandatory` | `Support`	Прикладное исключение	Если вызван `setRollbackOnly()`, помечает транзакцию для отката. Повторно возбуждает прикладное исключение.	Принимает прикладное исключение. Транзакция клиента может быть помечена для отката.
		Системное исключение	Помечает транзакцию клиента для отката. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `TransactionRolledbackException` для удаленного клиента или `javax.TransactionRolledbackLocalException` для локальных в EJB 2.0	Удаленные клиенты получают `TransactionRolledbackException`; локальные клиенты получают `javax.ejb.TransactionRolledbackException`. Выполняется откат транзакции клиента.

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Представление клиента
Транзакция, инициируемая контейнером. Транзакция начинается вызовом метода компонента и завершается по его окончании.	Тип транзакции = Контейнерная Атрибут транзакции = `Require` | `RequiresNew`	Прикладное исключение	Если вызывается `setRollbackOnly()` компонента, выполняет откат транзакции и повторно возбуждает прикладное исключение. Если компонент не выполняет явный откат транзакции, пытается зафиксировать транзакцию и повторно возбудить прикладное исключение.	Получает прикладное исключение. Может быть выполнен откат транзакции компонента. Транзакция клиента не затрагивается.
		Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBException` — для локальных в EJB 2.0	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBException`. Будет выполнен откат транзакции компонента. Транзакция клиента может быть помечена для отката в зависимости от производителя.

Таблица 14.1 (продолжение)

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Представление клиента
Компонент не является частью транзакции. Компонент вызывается, но не распространяет клиентскую транзакцию и не начинает свою.	Тип транзакции = Контейнерная Атрибут транзакции = `Never | NotSupported | Supports`	Прикладное исключение	Повторно возбуждает прикладное исключение.	Получает прикладное исключение. Клиентская транзакция не затрагивается.
		Системное исключение	Регистрирует ошибку. Удаляет компонент. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBExeption` для локальных клиентов в EJB 2.0.	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBExeption`. Клиентские транзакции могут быть помечены для отката в зависимости от производителя.
Транзакция, управляемая компонентом. Сеансовые компоненты с состоянием и без состояния используют `EJBContext` для явного управления своими транзакциями.	Тип транзакции = Компонентная Атрибут транзакции = Компоненты, управляющие транзакциями, не используют атрибуты транзакции.	Прикладное исключение	Повторно возбуждает прикладное исключение.	Получает прикладное исключение. Клиентския транзакция не затрагивается.

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Представление клиента
		Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBExeption` для локальных клиентов в EJB 2.0.	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBException`. Клиентская транзакция не затрагивается.

В табл. 14.2 представлен краткий обзор отношений между разными типами исключений и транзакций в компонентах, управляемых сообщениями.

Таблица 14.2. Обзор исключений в компонентах, управляемых сообщениями

Зона транзакции	Атрибуты транзакции	Возбуждаемое исключение	Действия контейнера
Транзакция, инициируемая контейнером. Транзакция начинается перед вызовом метода `onMessage()` и заканчивается по его завершении.	Тип транзакции = Контейнерная Атрибут транзакции = `Required`	Системное исключение	Регистрирует ошибку. Удаляет экземпляр.
Транзакция, инициируемая компонентом.	Тип транзакции = Контейнерная Атрибут транзакции = `NotSupported`	Системное исключение	Регистрирует ошибку. Удаляет компонент.
Транзакция, управляемая компонентом. Компонент для явного управления своей транзакцией использует `EJBContext`.	Тип транзакции = Компонентная Атрибут транзакции = Компонент, управляющий транзакцией, не использует атрибуты транзакции	Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет компонент.

Транзакционные сеансовые компоненты с состоянием

Как было показано в главе 12, сеансовые компоненты могут взаимодействовать непосредственно с базой данных так же легко, как они могут управлять рабочим потоком других компонентов. Компонент ProcessPayment, например, делает вставки в таблицу PAYMENT, когда вызывается метод byCredit(), а компонент TravelAgent генерирует запрос непосредственно к базе данных, когда вызывается метод listAvailableCabins(). Сеансовые компоненты без состояния, такие как ProcessPayment, не имеют никакого состояния диалога, поэтому каждый вызов метода должен сразу же вносить изменения в базу данных. Однако в случае с сеансовыми компонентами с состоянием мы не должны вносить изменения в базу данных до завершения транзакции. Запомните, что сеансовый компонент с состоянием может быть единственным участником многих транзакций, поэтому желательно отложить обновление базы данных до закрепления всей транзакции или обойтись без обновлений, если выполняется ее откат.

Существует несколько различных сценариев, в которых сеансовому компоненту с состоянием требуется кэшировать изменения перед закреплением их в базе данных. Представьте себе корзину покупателя, реализуемую сеансовым компонентом с состоянием, который накапливает покупаемые товары. Если компонент с состоянием реализует интерфейс SessionSynchronization, он может кэшировать элементы и записывать их в базу данных только по завершении транзакции.

Интерфейс javax.ejb.SessionSynchronization позволяет сеансовому компоненту принимать дополнительное уведомление о включении его сессии в транзакцию. Добавление интерфейсом SessionSynchronization этих транзакционных методов обратного вызова расширяет жизненный цикл компонента, добавляя в него новое состояние – *транзакционное состояние готовых методов* (*Transactional Method-Ready state*). Это третье состояние, хотя оно и не рассматривалось в главе 12, всегда является частью жизненного цикла транзакционного сеансового компонента с состоянием. Реализация интерфейса SessionSynchronization просто делает его видимым компоненту. На рис. 14.11 показан сеансовый компонент с этим дополнительным состоянием.

Интерфейс SessionSynchronization определен следующим образом:

```
package javax.ejb;

public interface javax.ejb.SessionSynchronization {
    public abstract void afterBegin() throws RemoteException;
    public abstract void beforeCompletion() throws RemoteException;
    public abstract void afterCompletion(boolean committed) throws
RemoteException;
}
```

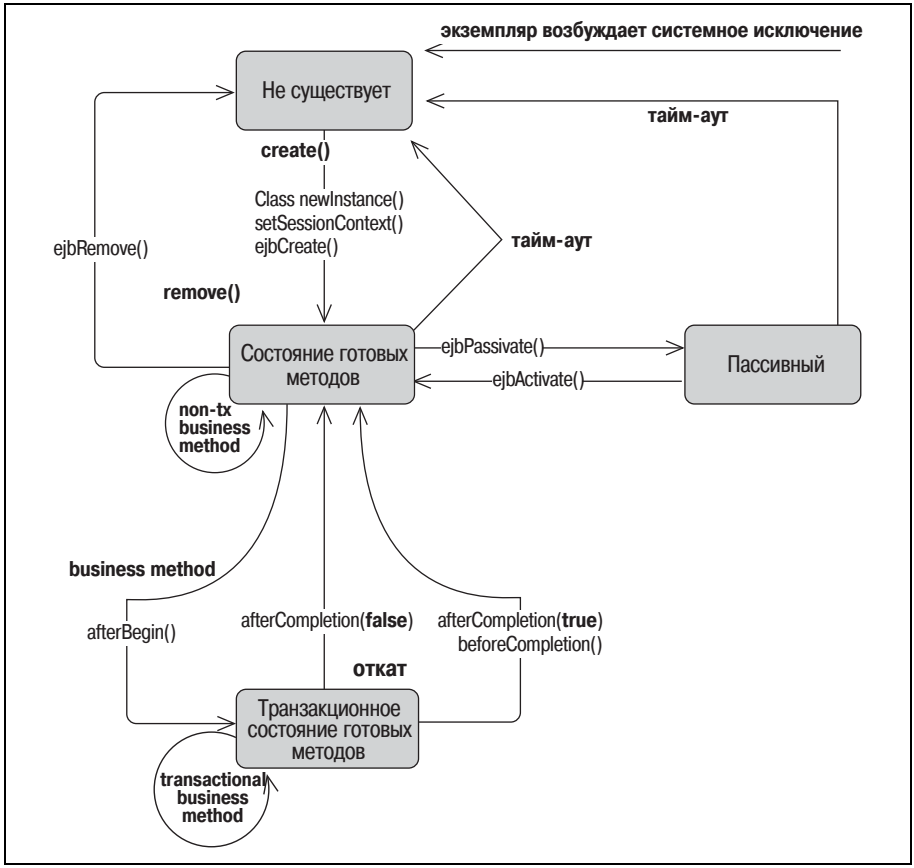

Рис. 14.11. Жизненный цикл сеансового компонента с состоянием

Если метод компонента `SessionSynchronization` вызывается извне зоны транзакции, этот метод выполняется в состоянии готовых методов, как было рассмотрено в главе 12. Однако если метод вызывается изнутри зоны транзакции (или создает новую транзакцию), компонент перемещается в транзакционное состояние готовых методов.

Транзакционное состояние готовых методов

Методы `SessionSynchronization` вызываются в транзакционном состоянии готовых методов.

Переход в транзакционное состояние готовых методов

Когда вызывается транзакционный метод компонента `SessionSynchronization`, компонент с состоянием становится частью транзакции. Это приводит к вызову метода обратного вызова `afterBegin()`, определенного в интерфейсе `SessionSynchronization`. Данный метод должен позабо-

титься о чтении из базы данных любых данных и сохранении этих данных в полях экземпляра компонента. Метод `afterBegin()` вызывается перед тем, как компонентный объект передаст экземпляру компонента вызов прикладного метода.

Жизнь в транзакционном состоянии готовых методов

После завершения метода обратного вызова `afterBegin()` данным экземпляром будет выполнен прикладной метод, первоначально вызванный клиентом. Все последующие прикладные методы, вызываемые изнутри этой же транзакции, будут перенаправляться непосредственно экземпляру EJB.

После того как сеансовый компонент с состоянием станет частью транзакции, независимо от того, реализует ли он `SessionSynchronization` или нет, к нему не может обращаться никакой другой транзакционный контекст. Это справедливо независимо от того, пытается ли клиент обращаться к компоненту из другого контекста, или собственный метод компонента создает новый контекст. Если, например, вызывается метод с атрибутом транзакции `RequiresNew`, то новый транзакционный контекст вызывает генерацию ошибки. Поскольку атрибуты `NotSupported` и `Never` задают другой транзакционный контекст (без контекста), вызов метода с этими атрибутами также приводит к ошибке. Сеансовый компонент с состоянием не может быть удален, пока он связан с транзакцией. Это означает, что вызов `ejbRemove()` в то время, когда компонент `SessionSynchronization` находится в середине транзакции, вызовет возбуждение ошибки.

В некоторой точке транзакция, в которой участвует компонент `SessionSynchronization`, достигнет конца. Если транзакция фиксируется, компонент `SessionSynchronization` будет уведомлен об этом через свой метод `beforeCompletion()`. В это время компонент должен записать его кэшированные данные в базу данных. Если выполняется откат транзакции, то метод `beforeCompletion()` не будет вызван во избежание ненужных затрат, связанных с записью изменений, которые не будут закреплены в базе данных.

Метод `afterCompletion()` вызывается всегда, независимо от результата завершения транзакции. Если транзакция закончилась успешно, т. е. был выполнен вызов `beforeCompletion()`, то параметр `committed` метода `afterCompletion()` будет установлен в `true`. Если транзакция была неудачна, `committed` будет иметь значение `false`.

Если метод `afterCompletion()` указывает на то, что транзакция была отменена, то, возможно, потребуется установить переменные экземпляра сеансового компонента с состоянием в некоторое начальное состояние.

15

Стратегии дизайна

В предыдущих четырнадцати главах мы познакомились с основными технологиями EJB. Осталось рассмотреть кое-какие вопросы, оставленные ранее за кадром: решение частных задач проектирования, работу с отдельными видами баз данных и другие подобные темы.

Хеш-коды в составных первичных ключах

В главе 11 обсуждалась необходимость перегрузки (overriding) методов `Object.hashCode()` и `Object.equals()` класса первичного ключа объектного компонента. В случае со сложными первичными ключами, имеющими несколько полей, перегрузка `Object.equals()` достаточно тривиальна. Однако перегрузка `Object.hashCode()` более сложна, поскольку целочисленное значение, которое может служить подходящим хеш-кодом, должно быть собрано из нескольких полей.

Одно решение состоит в том, чтобы объединить все значения в строку и использовать метод `hashCode()` объекта `String` для создания значения хеш-кода для всего первичного ключа. Класс `String` имеет достаточно эффективный алгоритм создания хеш-кода, генерирующий довольно хорошо распределенные и повторяющиеся значения хеш-кодов из любого набора символов.

В следующем коде показано, как создать такой хеш-код для гипотетического первичного ключа:

```
public class HypotheticalPrimaryKey implements java.io.Serializable {
    public int primary_id;
```

```
    public short secondary_id;
    public java.util.Date date;
    public String desc;

    public int hashCode() {

        StringBuffer strBuff = new StringBuffer();
        strBuff.append(primary_id);
        strBuff.append(secondary_id);
        strBuff.append(date);
        strBuff.append(desc);
        String str = strBuff.toString();
        int hashCode = str.hashCode();
        return hashCode;
    }
    // Далее идут конструктор и методы equals и toString
}
```

При создании нескольких объектов применение `StringBuffer` более эффективно, поскольку объединение объектов `String` работает довольно медленно. Данный код можно улучшить с помощью сохранения хеш-кодов в закрытой переменной и возвращения этого значения при последующих вызовах методов. Таким образом, хеш-код рассчитывается только один раз за время жизни экземпляра.

Хорошо распределенные и уникальные хеш-коды

Объект `Hashtable` предназначен для реализации быстрого поиска с помощью привязки объекта к ключу. Имея ключ какого-либо объекта, можно достаточно быстро найти объект в хеш-таблице. Для этого поиска ключ преобразуется в целочисленное значение при помощи метода ключа `hashCode()`.

Хеш-коды не обязаны быть уникальными, а только хорошо распределенными. Говоря «хорошо распределенный», мы подразумеваем, что при наличии любых двух ключей шансы на то, что они будут иметь разные хеш-коды, очень велики. Алгоритм хорошо распределенных хеш-кодов уменьшает, но не исключает возможность вычисления одинаковых хеш-кодов для разных ключей. Если для ключей вычисляется один и тот же хеш-код, они сохраняются вместе и уникально идентифицируются своими методами `equals()`. Если для поиска объекта используется ключ, для которого вычислен хеш-код, совпадающий с несколькими другими ключами, `Hashtable` находит группу объектов, сохраненных с одним и тем же хеш-кодом. Затем он вызывает метод ключа `equals()` для того, чтобы определить, какой ключ (а следовательно, и какой объект) нам нужен. (Именно поэтому мы должны перегружать (override) метод `equals()` первичных ключей, как и метод `hashCode()`.) Следовательно, основное внимание при проектировании хорошего алгоритма хеш-кодов следует уделять созданию не уникальных, а хорошо распределенных кодов. Эта стратегия позволяет проек-

тировать легко вычисляемый и, следовательно, быстрый индекс для связывания ключей с объектами.

Передача объектов по значению

Передача объектов по значению в Enterprise JavaBeans достаточно хитроумна. Сформулируем два простых правила, соблюдение которых оградит вас от большинства проблем: объекты, передаваемые по значению, должны быть небольшими зависимыми объектами или обертками, используемыми в методах пакетного доступа (bulk accessor); и зависимые объекты должны быть неизменяемыми (immutable).

EJB 1.1: Зависимые объекты

Понятие зависимых объектов было введено в главе 6, в которой описывалось применение зависимых объектов в EJB 2.0. Но для EJB 1.1 зависимые объекты представляют собой новое понятие. EJB 2.0 и EJB 1.1 используют зависимые объекты по-разному, т. к. EJB 2.0 может работать с гораздо более мелкими объектными компонентами, чем EJB 1.1.

Зависимые объекты – это объекты, имеющие смысл лишь внутри контекста другого прикладного объекта. Обычно они представляют достаточно небольшие прикладные понятия, такие как адреса, номера телефона или пункт заказа. Например, адрес имеет мало смысла, если он не связан с прикладным объектом типа Person (Человек) или Organization (Организация). Его значение зависит от контекста прикладного объекта. Такой объект можно представить себе как обертку для связанных данных. Поля, из которых состоит адрес (`street`, `city`, `state` и `zip`), должны быть упакованы вместе в одном объекте с именем `AddressDO`. В свою очередь, объект `AddressDO` – это обычно атрибут или свойство другого прикладного объекта. В EJB мы обычно встречаем зависимые объекты типа `AddressDO` в виде свойства объектного компонента.

Далее приводится типичная реализация объекта `AddressDO`:

```
public class AddressDO implements java.io.Serializable {

    private String street;
    private String city;
    private String state;
    private String zip;

    public Address(String str, String cty, String st, String zp) {
        street = str;
        city = cty;
        state = st;
        zip = zp;
    }
    public String getStreet() {return street;}
```

```
    public String getCity() {return city;}
    public String getState() {return state;}
    public String getZip() {return zip;}
}
```

Мы хотим быть уверенными, что клиенты не изменят поля объекта AddressDO. Причина весьма проста: объект AddressDO представляет собой копию, а не удаленную ссылку. Изменения объекта AddressDO не будут отражены в объекте, создавшем его. Если клиент изменит объект AddressDO, то изменения не будут отражены в базе данных. Сделав объект AddressDO неизменяемым, мы гарантируем, что клиенты не примут этот небольшой объект за удаленную ссылку и не подумают, что изменение его свойства Address отразится на сервере.

Для того чтобы изменить адрес, клиенту потребуется удалить объект AddressDO и добавить новый, учтя эти изменения. И снова это требуется потому, что зависимый объект – это не удаленный объект и изменение его состояния не отразится на сервере. Далее приведен удаленный интерфейс для гипотетического компонента Employee (Служащий), который включает информацию об адресе:

```
public interface Employee extends javax.ejb.EJBObject {
    public AddressDO [] getAddresses() throws RemoteException;
    public void removeAddress(AddressDO adrs) throws RemoteException;
    public void addAddress(AddressDO adrs) throws RemoteException;
    // ... Другие прикладные методы
}
```

В этом интерфейсе служащий может иметь несколько адресов, доступных в виде коллекции объектов AddressDO, передаваемых по значению. Чтобы удалить адрес, целевой объект AddressDO передается обратно компоненту через метод removeAddress(). После этого класс компонента удаляет соответствующий объект AddressDO из своих полей постоянства. Для того чтобы добавить адрес, объект AddressDO передается компоненту по значению.

Зависимые объекты могут представлять собой поля постоянства или свойства, создаваемые при необходимости. В следующем коде показаны обе стратегии использования объекта AddressDO. В первом листинге объект AddressDO является полем постоянства, тогда как во втором – свойством, не соответствующим никакому отдельному полю. Мы создаем объект AddressDO при необходимости, но не храним его в виде части компонента. Вместо этого объект AddressDO соответствует четырем полям постоянства: street, city, state и zip.

```
// Адрес в виде постоянного поля
public class Person extends javax.ejb.EntityBean {
    public AddressDO address;
    public AddressDO getAddress() {
        return address;
```

```
    }
    public void setAddress(AddressDO addr) {
        address = addr;
    }
    ...
}

// Адрес в виде свойства
public class Person extends javax.ejb.EntityBean {
    public String street;
    public String city;
    public String state;
    public String zip;

    public AddressDO getAddress() {
        return new AddressDO(street, city, state, zip);
    }
    public void setAddress(AddressDO addr) {
        street = addr.street;
        city = addr.city;
        state = addr.state;
        zip = addr.zip;
    }
    ...
}
```

Зависимый объект, выступающий в качестве свойства, может синхронизироваться с полями постоянства непосредственно в методах доступах или в методах `ejbLoad()` и `ejbStore()`. Обе стратегии приемлемы.

Настоящее обсуждение зависимых объектов изобилует обобщениями и, следовательно, не может применяться во всех ситуациях. Так, мы говорим, что по значению рекомендуется передавать только очень мелкие, зависимые, неизменяемые объекты. Все остальные прикладные понятия должны быть представлены в виде объектных или сеансовых компонентов. Очень мелкий объект – это тот, который характеризуется очень простым поведением и состоит главным образом из методов set и get. Зависимый объект – это тот, который не имеет большого смысла вне контекста агрегирующего объекта. Неизменяемый объект – это тот, который предоставляет только методы get и, следовательно, не может быть изменен после создания.

Правила проверки корректности в зависимых объектах

Зависимые объекты прекрасно подходят для использования правил проверки формата. Проверка формата гарантирует, что простая конструкция данных удовлетворяет предопределенной структуре или форме. В качестве примера возьмем почтовый индекс, всегда имеющий определенный формат. Он должен состоять из цифр, он должен иметь

длину в пять или девять цифр, и если он имеет девять цифр, то в нем пятая и шестая цифры разделяются при помощи дефиса. Проверка соответствия почтового индекса (в данном случае) этим правилам называется проверкой корректности формата.

Проблема, с которой сталкиваются все разработчики, состоит в том, чтобы определить, где следует поместить код проверки корректности. Надо ли проверять данные в пользовательском интерфейсе (ПИ), или это будет делать компонент, использующий данные? Проверка данных в ПИ имеет то преимущество, что она экономит сетевые ресурсы и улучшает производительность. Однако проверка данных в компоненте, расположенном на промежуточном уровне, гарантирует, что к его логике будут многократно обращаться разные пользовательские интерфейсы. Зависимые объекты представляют собой логичный компромисс, который позволяет данным быть проверенными на стороне клиента, но оставаться независимыми от ПИ. Поскольку логика проверки расположена в конструкторе зависимого объекта, объект автоматически проверяет данные при своем создании. Когда данные передаются в ПИ (например, GUI, сервлет или JSP), ПИ может проверить их с помощью своего соответствующего зависимого объекта. Если данные являются правильными, зависимый объект создается. Если же данные – неверные, конструктор генерирует исключение.

В следующем коде показан зависимый объект, который представляет почтовый индекс. Он придерживается сформулированных здесь правил для зависимого объекта, а также содержит в конструкторе правила проверки формата:

```
public class ZipCodeDO implements java.io.Serializable {

    private String code;
    private String boxNumber;

    public ZipCode(String zipcode) throws ValidationException {
        if (zipcode == null)
            throw new ValidationException("Индекс не может быть null");
        else if (zipcode.length()==5 && ! isDigits(zipcode))
      throw new ValidationException("Индекс должен состоять только из цифр");
        else if (zipcode.length()==10 ) {
            if (zipcode.charAt(5) == '-' ) {
                code = zipcode.substring(0,5);
                if (isDigits( code )){
                    boxNumber = zipcode.substring(6);
                    if (isDigits( boxNumber ))
                        return;
                }
            }
         throw new ValidationException("Индекс должен быть введен в форме
            #####-####");
        }
    }
```

```
    private boolean isDigits(String str) {
        for (int i = 0; i < str.length(); i++) {
            char chr = str.charAt(i);
            if ( ! Character.isDigit(chr)) {
                return false;
            }
        }
        return true;
    }
    public String getCode() { return code; }

    public String getBoxNumber() { return boxNumber; }

    public String toString() {
        return code+'-'+boxNumber;
    }
}
```

Этот простой пример иллюстрирует, что проверка корректности формата может выполняться зависимым объектом, когда он создается в интерфейсе пользователя или на клиенте. Обо всех ошибках проверки корректности формата сообщается немедленно, без всякого взаимодействия с промежуточным уровнем приложения. Кроме того, любой прикладной объект, использующий `ZipCodeDO`, автоматически получает в свое распоряжение код проверки корректности, делая правила проверки корректности многократно используемыми (и согласованными) многими компонентами. Помещение проверки формата в зависимый объект следует считать хорошей практикой программирования, т. к. оно делает зависимый объект ответственным за проверку своей собственной корректности. А ответственность – это ключевая концепция в объектно-ориентированном программировании. Конечно, зависимые объекты полезны для проверки корректности, только если реализация Enterprise JavaBeans поддерживает передачу по значению.

В качестве альтернативы применению зависимых объектов проверка формата может быть выполнена методами доступа компонентов. Если, например, клиентский компонент имеет методы доступа для установки и получения почтового индекса, то они могут заключать в себе код для проверки корректности. Хотя эта стратегия более эффективна с точки зрения использования сети (передача значения `String` более эффективна, чем передача по значению зависимого объекта), многократное использование в ней реализуется хуже, чем при размещении правил проверки формата в зависимых объектах.

Методы пакетного доступа для удаленных клиентов

Большинство объектных компонентов имеют несколько полей постоянства, которые управляются через методы доступа. К сожалению, принцип «один-к-одному», реализуемый методами доступа, может

привести к многочисленным вызовам при обращениях к объекту, которые при использовании удаленных ссылок вызовут резкое увеличение сетевого трафика. Каждое поле, которое необходимо изменить, потребует вызова метода, что, в свою очередь, потребует выхода в сеть. Один из способов уменьшить сетевой трафик при доступе к объектам удаленных клиентов состоит в применении пакетных методов доступа (bulk accessor), упаковывающих обращение к нескольким полям постоянства в один метод доступа. Пакетные методы доступа предоставляют методы get и set, работающие со структурами и простыми объектами, передаваемыми по значению. В следующем коде показано, как пакетный метод доступа может быть реализован для компонента Cabin:

```
// CabinData DataObject
public class CabinData {
    public String name;
    public int deckLevel;
    public int bedCount;
    public CabinData() {
    }
    public CabinData(String name, int deckLevel, int bedCount) {
        this.name = name;
        this.deckLevel = deckLevel;
        this.bedCount = bedCount;
    }
}

// CabinBean с методами пакетного доступа
public class CabinBean implements javax.ejb.EntityBean {
    public int id;
    public String name;
    public int deckLevel;
    public int ship;
    public int bedCount;
    // Методы пакетного доступа
    public CabinData getData() {
        return new CabinData(name,deckLevel,bedCount);
    }
    public void setData(CabinData data) {
        name = data.name;
        deckLevel = data.deckLevel;
        bedCount = data.bedCount;
    }
    // Простые методы доступа
    public String getName() {
        return name;
    }
    public void setName(String str) {
        name = str;
    }
```

```
    // Другие методы
}
```

Методы `getData()` и `setData()` позволяют упаковать несколько полей в простой объект и передавать их между клиентом и компонентом за один вызов метода. Это гораздо эффективнее выполнения трех отдельных вызовов для установки названия, уровня палубы и количества спальных мест.

Правила применения пакетных методов доступа

Здесь приведено несколько рекомендаций по созданию пакетных методов доступа.

Объекты с данными не являются зависимыми объектами

Объекты с данными (entity objects) и зависимые объекты служат совершенно разным целям, но на первый взгляд они могут показаться очень похожими. Тогда как зависимые объекты представляют прикладные понятия, объекты с данными этого не делают. Они просто обеспечивают эффективный способ упаковки полей объекта с целью доступа к ним клиентов. Объекты с данными могут упаковывать зависимые объекты вместе с большинством примитивных атрибутов, но сами они не являются зависимыми объектами.

Объекты с данными – это простые структуры

Делайте объекты с данными настолько простыми, насколько это возможно. В идеале они должны быть похожи на простую структуру языка C. Другими словами, объект с данными не должен иметь никакой прикладной логики вообще, у него должны быть только поля. Вся прикладная логика должна быть сосредоточена в объектном компоненте, где она сконцентрирована и может легко управляться.

Для сохранения синтаксиса структуры языка C объекты с данными не должны иметь методов доступа (get и set) для чтения и записи своих полей. Класс `CabinData` не содержит методов доступа, у него есть только поля и пара конструкторов. Отсутствие методов доступа подчеркивает, что объект с данными существует только для связывания полей вместе, а не для того чтобы действовать определенным образом. Принцип проектирования состоит в том, что объект с данными должен быть простой структурой, лишенной какого-либо поведения, другими словами, форма определяется назначением. Исключением является конструктор с параметрами, который оставлен для удобства разработчика.

Методы пакетного доступа объединяют связанные поля

Методы пакетного доступа могут передавать подмножество данных объекта. Некоторые поля могут иметь разную степень безопасности или потребности в транзакциях, а это требует, чтобы к ним обрати-

лись отдельно. В CabinBean в объект с данными передается единственное подмножество полей (name, deckLevel, bedCount). Поле id не было включено по нескольким причинам: оно не описывает прикладное понятие, оно уже находится в первичном ключе, и клиент не должен его модифицировать. Поле ship не передается, потому что его должны модифицировать только некоторые лица. Лица, которым разрешается модифицировать данное поле, отличаются от тех, которым разрешено изменять другие поля. Точно так же доступ к ship может подпадать под уровень изоляции транзакций, отличный от уровня изоляции, принятого для других полей (например, сериализуемый и закрепленное чтение).

Кроме того, большую эффективность обеспечивают методы пакетного доступа, которые передают логически связанные поля. В объектных компонентах с многими полями можно группировать вместе некоторые поля, которые обычно редактируются одновременно. Компонент Employee, например, мог бы включать в себя несколько полей, которые по своей природе являются демографическими (address, phone, email) и могут быть логически отделены от полей, которые отведены для пособий (compensation, 401K, health, vacation). Группа логически связанных полей может иметь свой собственный метод пакетного доступа. Для одного компонента могут даже понадобиться несколько методов пакетного доступа:

```
public interface Employee extends javax.ejb.EJBObject {

    public EmployeeBenefitsData getBenefitsData()
        throws RemoteException;

    public void setBenefitsData(EmployeeBenefitsData data)
        throws RemoteException;

    public EmployeeDemographicData getDemographicData()
        throws RemoteException;

    public void setDemographicData(EmployeeDemographicData data)
        throws RemoteException;

    // Другие методы доступа и прикладные методы
```

Не усложняйте методы доступа

Применяя методы пакетного доступа, не следует отказываться от простых методов доступа (методы get и set для одиночных полей). Все так же важно разрешить изменение одиночных полей. Очень расточительно использовать метод пакетного доступа для изменения одного поля, как и изменять несколько полей с помощью простых методов доступа.

Локальные ссылки в постоянстве, управляемом контейнером, в EJB 2.0 весьма эффективны, поэтому выигрыш от применения методов пакетного доступа минимален. Если вы работаете с EJB 2.0, применяйте методы пакетного доступа в удаленных интерфейсах тогда, когда это имеет смысл, учитывая приведенные здесь рекомендации, но в локальных интерфейсах их использование лучше ограничить.

Объекты с данными

В предыдущем разделе, посвященном передаче объектов по значению, мы сформулировали некоторые базовые правила, определяющие способы и обстоятельства использования в EJB передачи по значению. Прикладные понятия, которые не удовлетворяют критериям зависимых объектов, должны моделироваться либо как сеансовые компоненты, либо как объектные компоненты. Достаточно просто по ошибке применить стратегию передачи по значению для прикладных объектов, обычно определяемых в виде объектных компонентов (например, Customer, Ship и City). Злоупотребление методами пакетного доступа для передачи объектов с данными, несущих в себе прикладное поведение, – плохой стиль проектирования. Идея в том, что такая передача клиенту объектов с данными позволит избежать ненужного сетевого трафика, оставляя методы set и get локальными. А неувязка состоит в эквивалентности объектов. Предполагается, что объекты представляют действительные данные в базе данных, а это означает, что они используются совместно и всегда отражают текущее состояние данных. После того как объект размещается на клиенте, он больше не представляет эти данные. Клиент легко может уничтожить много черновых копий одного объекта, что приведет к несогласованной обработке и представлению данных.

Хотя методы set и get объектов с данными действительно могут вызвать увеличение сетевого трафика, реализация объектов, передаваемых по значению, вместо использования объектных компонентов не является выходом. Чрезмерной загрузки сети можно избежать, если придерживаться стратегии дизайна, используемой в этой книге: удаленные клиенты взаимодействуют в основном с сеансовыми компонентами, а не с объектными. Также можно значительно уменьшить сетевой трафик, применяя методы пакетного доступа, при условии, что эти методы будут передавать только структуры без прикладной логики. Наконец, старайтесь держать объектные компоненты на сервере включенными в рабочий поток, заданный сеансовыми компонентами. Это устраняет сетевой трафик, связанный с объектами, и в то же время гарантирует, что они всегда будут представлять верные данные.

Улучшение производительности с помощью сеансовых компонентов

Кроме определения взаимодействий между объектными компонентами и другими ресурсами (рабочий поток) сеансовые компоненты имеют еще одно существенное преимущество: они улучшают производительность. Рост производительности в случае применения сеансовых компонентов связан с понятием *степени модульности* (*granularity*). Степень модульности описывает размер области прикладного компонента, т. е. показывает, какая часть прикладного пространства охватывается компонентом. Область компонентов ограничена отдельным понятием и может влиять только на данные, связанные с этим понятием. Сеансовые компоненты представляют большие, крупные компоненты с областью, охватывающей несколько прикладных понятий – все прикладные понятия и процессы, необходимые компоненту для решения своей задачи. В распределенных прикладных вычислениях для того, чтобы обеспечить простой, однородный, многократно используемый и безопасный доступ к данным, мы полагаемся на небольшие компоненты, такие как объектные компоненты. Крупные прикладные компоненты, например сеансовые компоненты, занимаются взаимодействиями между объектами и прикладными процессами, охватывающими несколько объектов так, чтобы они могли многократно использоваться. При этом они улучшают производительность и клиента и сервера. Как правило, клиентские приложения должны выполнять большую часть своей работы с помощью крупных компонентов, таких как сеансовые компоненты, и ограничить непосредственное взаимодействие с объектными компонентами.

Для того чтобы понять, как сеансовые компоненты улучшают производительность, мы должны представить наиболее общие проблемы, созданные распределенными компонентными системами: сетевой трафик, время ожидания и потребление ресурсов.

Сетевой трафик и время ожидания удаленных клиентов

Одна из наиболее серьезных проблем распределенных компонентных систем состоит в том, что они создают насыщенный сетевой трафик. Это особенно справедливо для компонентных систем, полагающихся исключительно на прикладные компоненты, такие как компонент `EntityBean`. Каждый вызов метода удаленной ссылки начинает цикл удаленного обращения к методу, в котором информация пересылается из заглушки на сервер и обратно. Этот цикл требует, чтобы данные передавались клиенту и обратно, занимая полосу пропускания. Система заказа билетов для круизов «Титан» использует несколько объектных компонентов (таких как Ship, Cabin, Cruise и Customer). По мере того

как мы обходим эти небольшие компоненты, запрашивая информацию, обновляя их состояние и создавая новые компоненты, мы генерируем сетевой трафик, если обращаемся к компонентам из удаленных клиентов. Один клиент, скорее всего, не генерирует большой трафик, но если мы умножим его уровень трафика на несколько тысяч (а таково может быть количество клиентов), проблемы станут очевидны. В конечном счете тысячи клиентов породят такой трафик, что пострадает система в целом.

Другой аспект сетевой связи – это *время ожидания* (*latency*). Время ожидания – это задержка между моментом времени, когда мы начинаем выполнять команду, и временем, когда она завершается. При использовании компонентов всегда существует небольшая задержка из-за времени, необходимого для передачи запросов через сеть. Каждый вызов метода требует цикла RMI, который занимает некоторое время на прохождение от клиента на сервер и обратно. Клиент, использующий несколько компонентов, будет вынужден находиться в ожидании при каждом вызове метода. Суммируясь, эти задержки могут привести к очень медленной работе клиентов, которым потребуется несколько секунд, чтобы ответить на каждое действие пользователя.

Доступ со стороны клиента к крупным сеансовым компонентам вместо небольших объектных компонентов может существенно уменьшить роль пропускной способности сети и снизить время ожидания. В главе 12 мы создали для компонента TravelAgent метод `bookPassage()`. Метод `bookPassage()` включает в себя взаимодействия между объектными компонентами, которые в противном случае были бы размещены на стороне клиента. За сетевую стоимость одного вызова метода на клиенте (`bookPassage()`) на сервере EJB выполняется сразу несколько задач. Применение сеансовых компонентов для инкапсуляции нескольких задач уменьшает число удаленных вызовов методов, необходимых для выполнения задачи, что приводит к уменьшению сетевого трафика и времени ожидания, возникающих при выполнении этих задач.

В EJB 2.0 хороший проект должен использовать удаленные компонентные интерфейсы сеансового компонента, который управляет рабочим потоком и локальные компонентные интерфейсы управляемых компонентов (и сеансовых и объектных). Это обеспечивает лучшую производительность.

Поддержание равновесия при использовании удаленных клиентов

Мы не хотим отказываться от прикладных объектных компонентов, поскольку они предоставляют несколько преимуществ перед традиционной двухуровневой моделью. Они позволяют нам инкапсулировать прикладную логику и данные прикладного понятия так, чтобы они могли применяться согласованно, многократно и безопасно многими

приложениями. Короче говоря, прикладные объектные компоненты лучше подходят для доступа к прикладному состоянию из-за того, что они упрощают доступ к данным.

Вместе с тем, мы не должны злоупотреблять объектными компонентами на удаленных клиентах. Вместо этого необходимо, чтобы клиент взаимодействовал с крупными сеансовыми компонентами, инкапсулирующими взаимодействия между небольшими объектными компонентами. Существуют ситуации, в которых клиентское приложение должно взаимодействовать с объектными компонентами непосредственно. Если удаленному клиентскому приложению требуется модифицировать определенный объект (например, изменить адрес клиента), предоставление клиенту объектного компонента более практично, чем использование сеансового компонента. Однако если должна быть выполнена задача, связанная с взаимодействиями между несколькими объектными компонентами (например, перевод денег с одного счета на другой), должен применяться сеансовый компонент.

Когда клиентскому приложению требуется выполнить над объектом определенную операцию, например обновление, имеет смысл сделать объект непосредственно доступным клиенту. Если клиент выполняет задачу, охватывающую несколько прикладных понятий или как-то иначе включающую несколько объектов, эту задачу лучше смоделировать сеансовым компонентом в виде рабочего потока. В хорошем проекте упор делается на применение крупных сеансовых компонентов в качестве рабочего потока, и ограничивается число операций, требующих от клиента прямого доступа к объектным компонентам.

В EJB 2.0 объектные компоненты, которыми пользуются и удаленные клиенты, и локальные компоненты, могут реализовывать и удаленные, и локальные компонентные интерфейсы. Методы, определенные в удаленных и локальных компонентных интерфейсах, не должны быть одинаковыми. В каждом из них следует определять методы, подходящие для клиентов, которые будут с ними работать. Например, удаленные интерфейсы могли бы интенсивнее использовать методы пакетного доступа, чем локальный интерфейс.

Реализация списка

Принимайте решение о том, следует ли обращаться к данным непосредственно или через объектные компоненты, с осторожностью. Реализация списка, специфичная для рабочего потока, может быть предоставлена посредством прямого доступа к данным из сеансового компонента. Методы, подобные `listAvailableCabins()` в компоненте TravelAgent, используют прямой доступ к данным, поскольку он менее требователен, чем поисковый метод в компоненте Cabin, возвращающий список компонентов Cabin. Каждый компонент, с которым имеет дело система, требует ресурсов. Избегая применения компонентов

там, где их преимущество сомнительно, можно улучшить производительность всей системы. CTM похож на мощный грузовик, а каждый прикладной компонент, которым он управляет, – на небольшой груз. Грузовик гораздо больше подходит для перевозки набора грузов, чем легкий транспорт, такой как велосипед, но укладка слишком большого количества единиц груза на грузовик сделает его таким же неэффективным, как велосипед. Если никакое транспортное средство не может двигаться, какое из них лучше?

В главе 12 в качестве примера метода, возвращающего список табличных данных, рассматривался метод `listAvailableCabins()` компонента TravelAgent. В данном разделе представлено несколько разных стратегий для реализации списка в ваших компонентах.

Табличные данные – это данные, выстроенные в строки и столбцы. Табличные данные часто применяются для того, чтобы дать пользователям приложения возможность выбирать или проверять данные в системе. Enterprise JavaBeans разрешает обращаться к поисковым методам для получения списка объектных компонентов, но этот механизм не решает всех проблем. Во многих обстоятельствах поисковые методы, возвращающие удаленные ссылки, являются достаточно тяжеловесным решением несложной проблемы. Для примера в табл. 15.1 показано расписание круизов.

Таблица 15.1. Гипотетическое расписание круизов

Идентификатор круиза	Порт захода	Прибытие	Отправление
233	Сан-Хуан	4 июня 2002	5 июня 2002
233	Аруба	7 июня 2002	8 июня 2002
233	Картахена	9 июня 2002	10 июня 2002
233	Острова Сан-Блас	11 июня 2002	12 июня 2002

Можно было бы создать объект `Port-Of-Call` (порт захода), представляющий каждый пункт назначения, а затем получить список пунктов назначения с помощью поискового метода, но такой подход был бы неоправданно сложным. Осознав, что данные не используются совместно и только в этом случае от них есть толк, мы скорее представим их в виде простого табличного списка.

В данном случае мы представим эти данные клиенту компонента в виде массива объектов `String` со значениями, отделенными друг от друга символьными разделителями. Здесь приведен прототип метода, применяемого для получения данных:

```
public interface Schedule implements javax.ejb.EJBObject {
    public String [] getSchedule(int ID) throws RemoteException;
}
```

А далее приводится структура значений `String`, возвращенных методом `getSchedule()`:

```
233; San Juan; June 4, 2002; June 5, 2002
233; Aruba; June 7, 2002; June 8, 2002
233; Cartegena; June 9, 2002; June 10, 2002
233; San Blas Islands; June 11, 2002; June 12, 2002
```

Эти данные также могли быть возвращены в виде многомерного массива строк, в котором каждый столбец представляет одно поле. Это определенно облегчило бы обращение к каждому элементу данных, но вместе с тем, и усложнило бы навигацию по ним.

Один из недостатков подхода с простым массивом в том, что массивы в Java ограничены единственным типом. Другими словами, все элементы массива должны иметь один тип. Здесь мы имеем дело с массивом `Strings` из-за того, что он имеет максимальную гибкость при представлении других типов данных. Мы также могли бы создать массив объектов или даже `Vector`, но в этом случае не будет никакой информации о типах ни во время выполнения, ни во время разработки.

Реализация списков в виде массивов структур

Вместо того чтобы возвращать простой массив, метод, реализующий определенный вид списка, может возвращать массив структур. Например, данные о расписании судов круиза, приведенные в табл. 15.1, могли быть возвращены в виде массива структур, представляющих расписание. Эти структуры представляют собой простые объекты Java без какого-либо поведения (т. е. без методов), передаваемые в массиве. Возможные определения структуры и компонентного интерфейса приведены ниже:

```
// Определение компонента, использующего эту структуру
public interface Schedule implements javax.ejb.EJBObject {
    public CruiseScheduleItem [] getSchedule(int ID)
        throws RemoteException;
}

// Определение структуры
public class CruiseScheduleItem {
    public int cruiseID;
    public String portName;
    public java.util.Date arrival;
    public java.util.Date departure;
}
```

Применение структур позволяет элементам данных иметь разные типы. Кроме того, структуры являются самоописательными: достаточно легко узнать структуру данных в табличном наборе, основываясь на определении ее класса.

Реализация списков в виде ResultSets

Более сложный и гибкий способ реализации списка состоит в том, чтобы предоставить реализацию «передача по значению» интерфейса `java.sql.ResultSet`. Хотя он определен в пакете JDBC (`java.sql`), интерфейс `ResultSet` семантически независим от реляционных баз данных. Он может использоваться для представления любого набора табличных данных. Интерфейс `ResultSet` знаком большинству корпоративных разработчиков на Java, поэтому он представляет собой превосходную конструкцию для использования при реализации списка. В случае применения стратегии `ResultSet` сигнатура метода `getSchedule()` будет следующая:

```
public interface Schedule implements javax.ejb.EJBObject {
    public ResultSet getSchedule(int cruiseID) throws RemoteException;
}
```

В некоторых случаях табличные данные, отображаемые на клиенте, могут быть сгенерированы с помощью стандартного SQL через драйвер JDBC. Если позволяют обстоятельства, можно выбрать между выполнением запроса в сеансовом компоненте или вернуть результирующий набор непосредственно клиенту через метод списка. Однако существует много ситуаций, когда возвращать `ResultSet`, исходящий непосредственно от драйвера JDBC, не требуется. `ResultSet` драйвера JDBC 1.x обычно связан непосредственно с базой данных, что увеличивает загрузку сети и выставляет ваш источник данных клиенту. В этих случаях мы можем реализовать свой собственный объект `ResultSet`, использующий для буферизации данных массивы или векторы. JDBC 2.0 предоставляет буферизованный `javax.sql.RowSet`, похожий на `ResultSet`, но передаваемый по значению и предоставляющий возможность обратной прокрутки. Можно использовать `RowSet`, но не предоставлять функциональность, позволяющую модифицировать результирующий набор. Обновление данных должно выполняться только методами компонента.

Иногда табличные данные исходят из нескольких источников данных или из нереляционных баз данных. В этих случаях можно запрашивать данные с помощью подходящих механизмов внутри компонента, реализующего список, а затем преобразовать данные под свою реализацию `ResultSet`. Независимо от источника данных, их все еще необходимо представлять в виде табличных данных, используя собственную реализацию интерфейса `ResultSet`.

Применение `ResultSet` имеет ряд достоинств и недостатков. Сначала преимущества:

Логичный интерфейс для разработчиков

`ResultSet` предоставляет согласованный интерфейс, знакомый разработчикам и не противоречащий различным реализациям списка.

Разработчикам не приходится изучать несколько разных конструкций для работы с табличными данными. Они применяют один и тот же интерфейс `ResultSet` для всех методов списка.

Логичный интерфейс для автоматизации

`ResultSet` предоставляет согласованный интерфейс, позволяющий программным алгоритмам оперировать данными независимо от их содержания. Можно создать генератор для создания HTML или таблицу GUI на базе любого набора результатов, реализуемого `ResultSet`.

Операции над метаданными

Интерфейс `ResultSet` определяет несколько методов для метаданных, предоставляющих разработчикам информацию времени выполнения, описывающую результирующий набор, с которым они работают.

Гибкость

Интерфейс `ResultSet` не зависит от содержания данных, что позволяет изменять схемы табличных наборов независимо от интерфейсов. Изменение схемы не требует изменений сигнатур методов, оперирующих списком.

А теперь о недостатках применения `ResultSet`:

Сложность

Стратегия интерфейса `ResultSet` намного сложнее, чем возвращение простого массива или массива структур. Обычно она требует от программиста разработки собственной реализации интерфейса `ResultSet`. Будучи должным образом разработанной, такая реализация может многократно использоваться всеми вашими методами списка, но ее создание все же потребует значительных усилий.

Сокрытие структуры во время разработки

Хотя `ResultSet` может с помощью метаданных описывать себя во время выполнения, он не может описывать себя во время разработки. В отличие от простого массива и массива структур, интерфейс `ResultSet` не предоставляет во время разработки никаких подсказок относительно структуры лежащих в основе данных. Во время выполнения метаданные доступны, но во время разработки для явного описания структуры данных необходима хорошая документация.

Компонентные адаптеры

Одна из наиболее неуклюжих сторон типов компонентных интерфейсов в EJB состоит в том, что в некоторых случаях методы обратного вызова никогда не используются или не нужны компоненту вообще.

Простой объектный компонент, управляемый контейнером, мог бы содержать пустые реализации своих методов `ejbLoad()`, `ejbStore()`, `ejbActivate()`, `ejbPassivate()` и даже `setEntityContext()`. Сеансовые компоненты без состояния представляют собою даже лучший пример лишних методов обратного вызова: они должны реализовывать методы `ejbActivate()` и `ejbPassivate()` даже при том, что эти методы никогда не вызываются!

Для того чтобы упростить вид определений классов компонентов, мы можем ввести *классы адаптеров* (*adapter class*), скрывающие методы обратного вызова, которые никогда не используются или имеют только минимальные реализации. Здесь приведен адаптер для объектного компонента, предоставляющий пустые реализации всех методов `EntityBean`:

```
public class EntityAdapter implements javax.ejb.EntityBean {
    public EntityContext ejbContext;

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}

    public void setEntityContext(EntityContext ctx) {
        ejbContext = ctx;
    }
    public void unsetEntityContext() {
        ejbContext = null;
    }
    public EntityContext getEJBContext() {
        return ejbContext;
    }
}
```

Мы позаботились о сохранении `EntityContext` для использования этим подклассом. Мы можем сделать это, поскольку большинство объектных компонентов реализуют методы, связанные с контекстом, точно так же. Мы используем класс адаптера просто для того, чтобы управлять этой логикой в наших подклассах.

Если нам потребуется какой-либо метод обратного вызова, мы можем просто заместить его методом класса компонента.

Мы можем создать аналогичный класс адаптера и для сеансовых компонентов без состояния:

```
public class SessionAdapter implements javax.ejb.SessionBean {
    public SessionContext ejbContext;

    public void ejbActivate() {}
```

```
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        ejbContext = ctx;
    }
    public SessionContext getEJBContext() {
        return ejbContext;
    }
}
```

Не используйте эти классы адаптеров, если требуется заместить больше чем один или два их метода. Если необходимо реализовать несколько методов обратного вызова, то отказ от класса адаптера сделает код более ясным. Кроме того, класс адаптера влияет на иерархию наследования класса компонента. Если вам позднее понадобится реализовать другой суперкласс – тот, который содержит прикладную логику, – вам потребуется изменить наследование класса.

Реализация обычного интерфейса

В этой книге не рекомендуется реализовывать удаленный интерфейс класса компонента, даже когда это немного усложняет обеспечение согласованности между прикладными методами, определенными в удаленном интерфейсе, и соответствующими методами класса компонента. Существует достаточно оснований для того, чтобы не реализовывать удаленный интерфейс класса компонента, но кроме этого надо, чтобы обычный интерфейс гарантировал, что класс компонента и удаленный интерфейс определяют одни и те же прикладные методы. В данном разделе описывается альтернативный подход, позволяющий гарантировать согласованность между классом компонента и удаленным интерфейсом при помощи обычного интерфейса.

Почему класс компонента не должен реализовывать удаленный интерфейс

Между прикладными методами, определенными в классе `ShipBean`, и соответствующими им прикладными методами, определенными в интерфейсе `ShipRemote`, не должно быть никаких различий, кроме отсутствия исключения `java.rmi.RemoteException`. EJB требует, чтобы сигнатуры методов были сопоставлены так, чтобы удаленный интерфейс мог точно представлять класс компонента на стороне клиента. Почему бы не реализовать удаленный интерфейс `com.titan.ShipRemote` в классе ShipBean для того, чтобы гарантировать, что данные методы правильно согласованы?

EJB разрешает классу компонента реализовывать свой удаленный интерфейс, но эта практика не рекомендуется по двум причинам. Во-первых, удаленный интерфейс в действительности представляет собой расширение интерфейса `javax.ejb.EJBObject`, о котором мы узнали из главы 5. Этот интерфейс определяет несколько методов, реализуемых контейнером EJB при развертывании компонента. Ниже приведено определение интерфейса `javax.ejb.EJBObject`:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean isIdentical(EJBObject obj);
    public abstract void remove();
}
```

Определенные здесь методы реализуются и поддерживаются компонентным объектом для использования клиентским программным обеспечением и не реализуются классом `javax.ejb.EntityBean`. Другими словами, эти методы предназначены для реализации удаленного интерфейса, а не экземпляра компонента. Экземпляр компонента реализует прикладные методы, определенные в удаленном интерфейсе, но он делает это косвенно. Компонентный объект принимает все вызовы методов, выполненные для удаленного интерфейса. Те из них, которые являются прикладными методами (такие как методы `getName()` и `setCapacity()` компонента Ship), передаются экземпляру компонента. Другие методы, определенные в `EJBObject`, обрабатываются контейнером и никогда не перенаправляются экземпляру компонента.

Просто ради интереса измените определение компонента `ShipBean` так, чтобы он реализовал интерфейс `ShipRemote`, как показано здесь:

```
public class ShipBean implements ShipRemote {
```

Перекомпилируя `ShipBean`, мы должны получить пять ошибок, утверждающих, что `ShipBean` необходимо объявить как `abstract`, поскольку он не реализует методы интерфейса `javax.ejb.EJBObject`. EJB разрешает реализовать удаленный интерфейс, но, поступая так, мы загромождаем определение класса компонента набором методов, не имеющих никакого отношения к его функциональности. Эти методы можно скрыть в классе адаптера, однако использование адаптера для методов, имеющих пустые реализации, – это одно, а применение адаптера для методов, которые вообще не должны содержаться в классе, – совершенно другое, причем это определенно плохая практика.

Другая причина, по которой компоненты не должны реализовать удаленный интерфейс, состоит в том, что клиент может являться приложением на удаленном компьютере или другим компонентом. Компо-

ненты в качестве клиентов весьма распространены. При вызове метода объекта вызывающий объект иногда передает сам себя в качестве одного из параметров.[1] В обычном программировании на Java для передачи объектом ссылки на себя применяется ключевое слово `this`. Однако в EJB клиентам (даже являющимся компонентами) разрешается взаимодействовать только с удаленными компонентными интерфейсами. Когда один компонент вызывает метод другого компонента, ему не разрешается передавать ссылку `this`. Вместо этого он должен получить удаленную ссылку на себя из своего контекста и передать эту ссылку. Тот факт, что класс компонента не реализует свой удаленный интерфейс, не дает нам передавать ссылку `this` и вынуждает получать ссылку на интерфейс из контекста. Класс компонента не будет компилироваться, если пытаться использовать `this` в качестве удаленной ссылки. Предположите, например, что `ShipBean` должен вызвать `someMethod(ShipRemote ship)`. Он не может просто вызывать `someMethod(this)`, т. к. `ShipBean` не реализует `ShipRemote`. Однако если бы экземпляр компонента реализовывал удаленный интерфейс, мы могли бы по ошибке передать ссылку на экземпляр компонента другому компоненту, используя ключевое слово `this`.

Компоненты всегда должны взаимодействовать с удаленными ссылками на другие компоненты так, чтобы вызовы методов перехватывались компонентными объектами. Запомните, что компонентные объекты применяют к вызовам методов безопасность, транзакции, параллелизм и другие ограничения системного уровня перед их перенаправлением экземпляру компонента. Для управления компонентом во время выполнения компонентный объект взаимодействует с контейнером.

Подходящий способ получения удаленной ссылки на компонент из класса компонента обеспечивает `EJBContext`. Приведем пример:

```
public class HypotheticalBean extends EntityBean {
    public EntityContext ejbContext;
    public void someMethod() throws RemoteException {

        Hypothetical mySelf = (Hypothetical) ejbContext.getEJBObject();

        // Делаем с удаленной ссылкой что-нибудь интересное
    }
    // Другие методы
}
```

[1] Это часто происходит при циклических вхождениях, когда вызываемому объекту требуется информация о вызывающем объекте. Циклические вхождения не рекомендуется использовать в EJB, поскольку они требуют реентерабельного программирования, которого следует избегать.

EJB 2.0: Почему класс компонента не должен реализовывать локальный интерфейс

В EJB 2.0 класс компонента не должен реализовывать локальный интерфейс по тем же причинам, по которым он не должен реализовывать удаленный интерфейс. А именно, экземпляр компонента должен был бы поддерживать методы интерфейса `javax.ejb.EJBLocalObject`, которые не относятся к классу компонента.

Альтернатива в EJB 1.1: Прикладной интерфейс

Хотя реализация классом компонента своего удаленного интерфейса для обеспечения согласованности определения прикладных методов нежелательна, мы можем определить промежуточный интерфейс, который будет использоваться и классом компонента, и удаленным интерфейсом. Будем называть этот промежуточный интерфейс *прикладным интерфейсом (business interface)*.

Следующий код содержит пример прикладного интерфейса с именем `ShipBusiness`, определенного для компонента Ship. Все прикладные методы, ранее определенные в интерфейсе `ShipRemote`, теперь определяются в интерфейсе `ShipBusiness`. Прикладной интерфейс определяет все прикладные методы, в том числе каждое исключение, которое будет возбуждаться удаленным интерфейсом во время выполнения:

```
package com.titan.ship;
import java.rmi.RemoteException;

public interface ShipBusiness {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

После того как прикладной интерфейс определен, он может быть расширен удаленным интерфейсом. Удаленный интерфейс расширяет и интерфейс `ShipBusiness`, и `EJBObject`, присваивая все прикладные методы и методы `EJBObject`, реализуемые контейнером во время развертывания:

```
package com.titan.ship;
import javax.ejb.EJBObject;

public interface ShipRemote extends ShipBusiness, javax.ejb.EJBObject {
}
```

И наконец, мы можем реализовать прикладной интерфейс в классе компонента, как и любой другой интерфейс:

```
public class ShipBean implements ShipBusiness, javax.ejb.EntityBean {
    public int id;
    public String name;
    public int capacity;
    public double tonnage;

    public String getName() {
        return name;
    }
    public void setName(String n) {
        name = n;
    }
    public void setCapacity(int cap) {
        capacity = cap;
    }
    public int getCapacity() {
        return capacity;
    }
    public double getTonnage() {
        return tonnage;
    }
    public void setTonnage(double tons) {
        tonnage = tons;
    }

    // Другие методы...
}
```

В данном случае мы решили не возбуждать исключение `RemoteException`. Классы, реализующие интерфейсы, могут не генерировать исключения, определенные в данных интерфейсах. Однако они не могут добавлять исключения. Именно поэтому прикладной интерфейс должен объявлять, что его методы генерируют `RemoteException` и все прикладные исключения. Удаленный интерфейс не должен изменять прикладное определение интерфейса. Класс компонента может не генерировать `RemoteException`, но он обязан генерировать все исключения, специфичные для данного приложения.

Прикладной интерфейс представляет собой легко реализуемую стратегию дизайна, упрощающую создание компонентов. Я бы рекомендовал вам в собственных проектах применять стратегию прикладных интерфейсов. Запомните, что не нужно передавать прикладной интерфейс в вызове метода. В качестве типов возвращаемых значений в параметрах метода всегда используйте удаленный интерфейс компонента.

Объектные компоненты без конструирующих методов

Если предполагается, что объектный компонент никогда не будет создаваться клиентом, то можно просто не реализовывать во внутреннем интерфейсе метод `create()`. Это означает, что рассматриваемый объект может быть доступен только с помощью методов `find()` внутреннего интерфейса. В «Титане» можно было бы реализовать эту стратегию в компонентах Ship так, чтобы новые компоненты Ship должны были создаваться прямой вставкой записи в базу данных, – привилегия, которой может обладать только администратор базы данных (они не захотели бы, чтобы какой-нибудь сошедший с ума туристический агент вставлял случайные суда в их круизы).

EJB 1.1: Инструментальные средства отображения объектов и отношений

Некоторые производители EJB предоставляют средства отображения объектов и отношений, способные с помощью мастеров создавать объектные представления реляционных баз данных, генерировать из объектов таблицы и связывать существующие объекты с существующими таблицами. Рассмотрение этих инструментальных средств выходит за рамки данной книги, т. к. по своему характеру они довольно нестандартны и, как правило, не могут применяться для генерации компонентов, способных использоваться на других серверах EJB. Другими словами, во многих случаях не исключено, что как только мы начнем полагаться на некоторое средство сопоставления для определения постоянства компонентов, мы, возможно, не сможем перенести наши компоненты на другой сервер EJB: определение компонента будет привязано к данному средству сопоставления.

Инструментальные средства сопоставления могут значительно повысить продуктивность разработчиков компонентов, но прежде чем применять какое-то средство, необходимо изучить детали, специфичные для данной его реализации. Если нам в дальнейшем понадобится перенести приложение на более крупный и быстрый сервер EJB, необходимо будет убедиться, что применяемое средство сопоставления поддерживается и другими серверами.

Некоторые продукты, выполняющие сопоставление объектов с отношениями, используют JDBC. TOPLINK от Object People и ROF от Watershed являются примерами этого типа продуктов. Данные программы предоставляют большую гибкость при связывании объектов с реляционными базами данных и не так зависят от сервера EJB. Однако для того чтобы с данными продуктами можно было работать, необхо-

димо, чтобы их поддерживали серверы EJB. Поэтому снова позволю себе предостеречь вас, взвесьте все «за» и «против», собираясь работать с этими продуктами.

Избегайте эмуляции объектных компонентов с помощью сеансовых компонентов

Сеансовые компоненты, реализующие интерфейс `SessionSynchronization` (рассмотренный в главе 14), могут эмулировать часть функциональных возможностей объектных компонентов, реализующих постоянство. Этот подход дает пару преимуществ. Во-первых, как и объектные компоненты, эти сеансовые компоненты могут представлять прикладные понятия. Во-вторых, исключается зависимость от специфичного для конкретного производителя инструментального средства отображения объектов на отношения.

К сожалению, сеансовые компоненты не были предназначены для представления данных непосредственно в базе данных, поэтому применение их в качестве замены объектных компонентов достаточно проблематично. Объектные компоненты хорошо выполняют свои обязанности из-за того, что они являются транзакционными объектами. Изменения атрибутов компонента автоматически отражаются в базе данных транзакционно безопасным способом. Это невозможно в сеансовых компонентах с состоянием, поскольку они хотя и знают о существовании транзакций, сами не являются транзакционными. Это различие достаточно тонкое, но оно важно. В отличие от объектных компонентов, сеансовые компоненты с состоянием не используются совместно. Не существует никакого управления параллелизмом в ситуациях, когда два клиента пытаются обратиться к одному компоненту в одно и то же время. В случае с сеансовыми компонентами с состоянием каждый клиент получает свой собственный экземпляр, поэтому возможна одновременная работа с множеством копий одного сеансового компонента, представляющих одни и те же данные. Изоляция, реализуемая базой данных, может предотвратить некоторые проблемы, но опасность получения и использования «грязных» данных остается высокой.

Еще одна проблема состоит в том, что сеансовые компоненты, эмулирующие объектные компоненты, не могут иметь в своих внутренних интерфейсах поисковые методы. Объектные компоненты поддерживают поисковые методы как удобный способ поиска данных. Поисковые методы могли бы быть помещены в удаленный интерфейс сеансового компонента, но это не будет согласовываться с компонентной моделью EJB. Кроме того, для обеспечения транзакционной безопасности сеансовый компонент с состоянием должен использовать интерфейс `SessionSynchronization`, требующий, чтобы это происходило только в рамках

клиентской транзакции. Дело в том, что методы, такие как `ejbCreate()` и `ejbRemove()`, не являются транзакционными. Кроме того, метод `ejbRemove()` выполняет совершенно разные функции в сеансовых компонентах и в объектных компонентах. Должен ли `ejbRemove()` завершать сеанс связи, удалять данные или делать и то и другое?

Положив на одну чашу весов все преимущества, а на другую – проблемы и риск потери согласованности данных, я рекомендовал бы избегать применения сеансовых компонентов с состоянием для эмуляции объектных компонентов.

Прямой доступ к базам данных из сеансовых компонентов

Не исключено, что наиболее простой и переносимый способ использования сервера, поддерживающего только сеансовые компоненты, состоит в непосредственном обращении к базе данных. Мы делали что-то похожее в компонентах ProcessPayment и TravelAgent в главе 12. И мы просто полнее реализуем эту возможность, когда применение объектных компонентов невозможно. Следующий код представляет собой пример метода `bookPassage()` компонента TravelAgent, написанного с применением прямого доступа к данным через JDBC вместо использования объектных компонентов:

```
public Ticket bookPassage(CreditCard card, double price)
    throws RemoteException, IncompleteConversationalState {
    if (customerID == 0 || cruiseID == 0 || cabinID == 0) {
        throw new IncompleteConversationalState();
    }
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = getConnection();

        // Вносим заказ
        ps = con.prepareStatement("insert into RESERVATION "+
            "(CUSTOMER_ID, CRUISE_ID, CABIN_ID, PRICE) values (?,?,?,?)");
        ps.setInt(1, customerID);
        ps.setInt(2, cruiseID);
        ps.setInt(3, cabinID);
        ps.setDouble(4, price);
        if (ps.executeUpdate() != 1) {
            throw new RemoteException ("Failed to add Reservation to
            database");
        }

        // Вносим оплату
        ps = con.prepareStatement("insert into PAYMENT "+
```

```
                "(CUSTOMER_ID, AMOUNT, TYPE, CREDIT_NUMBER, CREDIT_EXP_DATE) "+
                "values(?,?,?,?,?)");
            ps.setInt(1, customerID);
            ps.setDouble(2, price);
            ps.setString(3, card.type);
            ps.setLong(4, card.number);
            ps.setDate(5, new java.sql.Date(card.experation.getTime()));
            if (ps.executeUpdate() != 1) {
                throw new RemoteException ("Failed to add Reservation to
                database");
            }
            Ticket ticket = new Ticket(customerID,cruiseID,cabinID,price);
            return ticket;

        } catch (SQLException se) {
            throw new RemoteException (se.getMessage());
        }
        finally {
            try {
                if (ps != null) ps.close();
                if (con!= null) con.close();
            } catch(SQLException se){
                se.printStackTrace();
            }
        }
    }
```

Здесь нет ничего непонятного: мы просто переопределили компонент TravelAgent так, чтобы он работал непосредственно с данными через JDBC, а не использовал объектные компоненты. Этот метод транзакционно безопасен, т. к. исключение, возбужденное где-нибудь внутри данного метода, вызовет откат всех вставок, сделанных в базу данных. Все очень просто и понятно.

Идея, стоящая за этой стратегией, состоит в том, чтобы продолжить моделирование рабочего потока или процессов с помощью сеансовых компонентов. Компонент TravelAgent моделирует процесс создания заказа. Его состояние диалога может быть изменено в течение сеанса связи, и на основании этого состояния изменения могут быть безопасно внесены в базу данных.

Сопоставление объектов с отношениями предоставляет другой механизм для «прямого» доступа к данным из сеансового компонента с состоянием. Преимущество средств сопоставления объектов с отношениями состоит в том, что данные могут быть инкапсулированы в виде «объектоподобного» объектного компонента.[1] Так, подход с отображе-

[1] Достаточно неприглядно. Но именно так переводится вполне благозвучный термин «object-like entity bean». – *Примеч. науч. ред.*

нием объектов и отношений мог бы быть очень похожим на наш проект с объектным компонентом. Сопоставление объектов с отношениями таит в себе проблему. Дело в том, что большинство инструментальных средств не являются стандартными и не могут многократно применяться на разных серверах EJB. Другими словами, средства сопоставления объектов и отношений могут привязывать вас к одной марке серверов EJB. Однако эти инструментальные средства представляют собой подходящий, безопасный и эффективный механизм для получения прямого доступа к базам данных в случаях, когда объектные компоненты недоступны.

Избегайте соединения сеансовых компонентов с состоянием

При разработке систем, состоящих исключительно из сеансовых компонентов, может понадобиться использовать сеансовые компоненты с состоянием изнутри других сеансовых компонентов с состоянием. Хотя, на первый взгляд, данный подход кажется подходящим для моделирования, на самом деле он достаточно проблематичен. Объединение сеансовых компонентов с состоянием в цепочку может стать источником неприятностей в случае истечения времени жизни компонента или при возникновении исключения, приводящего к их недействительности. На рис. 15.1 показана цепочка сеансовых компонентов с состоянием, поддерживающих состояние диалога, от которого зависят другие компоненты при завершении операции, выполняемой компонентом A.

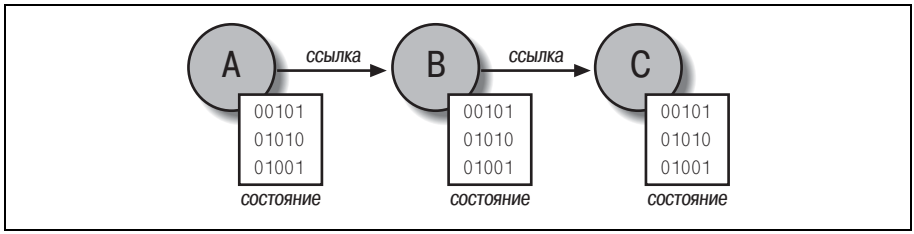

Рис. 15.1. Цепочка сеансовых компонентов с состоянием

Если у какого-либо из компонентов в цепочке закончится время жизни (истечет «тайм-аут») – скажем, у компонента B, – состояние диалога после этого компонента будет потеряно. Если это состояние диалога строилось на протяжении долгого времени, может быть потеряна значительная часть работы. Прочность цепочки сеансовых компонентов с состоянием определяется ее самым слабым звеном. Если один из компонентов прекращает существование или становится недействительным, становится недействительным все состояние диалога, зависящее

от этого компонента. Избегайте цепного объединения сеансовых компонентов с состоянием.

Обращение к сеансовым компонентам без состояния изнутри сеансовых компонентов с состоянием не вызывает проблем, поскольку сеансовый компонент без состояния не поддерживает никакого состояния диалога. Используйте сеансовые компоненты без состояния изнутри сеансовых компонентов с состоянием столько, сколько понадобится.

Обращение к сеансовому компоненту с состоянием из сеансового компонента без состояния почти не имеет смысла из-за того, что вне области метода сеансового компонента без состояния нельзя воспользоваться преимуществами состояния диалога сеансового компонента с состоянием.

16

XML-дескрипторы развертывания

Что такое XML-дескриптор развертывания?

В этой главе рассматривается состав XML-дескриптора развертывания. Здесь вы научитесь писать дескрипторы развертывания для своих компонентов. Имейте в виду, не исключено, что вам никогда не придется писать дескриптор развертывания вручную, большинство производителей интегрированных средств разработки и серверов EJB предоставляют средства для автоматического создания дескриптора. Однако даже имея доступ к такому средству, необходимо быть достаточно хорошо знакомым с дескрипторами развертывания, чтобы уметь читать их самостоятельно.

В этой главе не делается попыток научить вас читать или писать правильный XML. На эту тему существует немало книг: есть хороший краткий справочник «XML Pocket Reference» (Карманный справочник по XML) Боба Экштейна (Bob Ekstein, O'Reilly), а более подробную информацию можно найти в книге Эллиотта Расти Гарольда (Elliotte Rusty Harold) и У. Скотта Минса (W. Scott Means) «XML in a Nutshell», O'Reilly.[1] Если очень коротко, то XML напоминает HTML, но имена тегов и их атрибуты в нем другие. Внутри дескриптора развертывания не найти ни `<h1>`, ни `<p>`. Там есть другие теги, например `<ejb-jar>`. Что же

[1] Эллиотт Расти Гарольд, У. Скотт Минс «XML. Справочник», издательство «Символ-Плюс», 2001 г.

касается всего остального, тот, кто полагает, что XML-документ похож на HTML, находится на пути к тому, чтобы читать его. Имена тегов и атрибутов для XML-документа определены в специальном документе, называемом определением типа документа (Document Type Definition, DTD). Следовательно, для XML-дескрипторов развертывания существует DTD, в котором определены теги и атрибуты, которые могут фигурировать в документе. DTD для дескрипторов развертывания в EJB 2.0 и 1.1 доступны в Интернете по адресам: *http://java.sun.com/dtd/ejb-jar_2_0.dtd* и *http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd*.

Существует несколько других важных различий между XML и HTML. XML является гораздо более строгим. Многое из того, что допустимо в HTML, неприемлемо в XML. Это не должно иметь значения для тех, кто просто читаете дескриптор развертывания, но тот, кто его пишет, должен быть внимательным. Особенно важны следующие два различия. Во-первых, XML чувствителен к регистру. Нельзя смешивать верхний и нижний регистры в именах тегов. HTML не обращает внимания на различия между `<h1>` и `<H1>`, а XML заметит это. Все теги и атрибуты в дескрипторах развертывания записываются в нижнем регистре. Во-вторых, XML не простит отсутствие закрывающих тегов. В HTML допускается запись `<p>...<p>`, без включения тега `</p>`, завершающего первый абзац. XML не позволит автору быть небрежным. Всякий раз, записывая открывающий тег, необходимо вписать и закрывающий.

И это все. Эти несколько абзацев не могут считаться даже кратким введением в XML, но основные идеи очень просты, и это действительно все, что вам надо знать, чтобы двигаться дальше.

Содержимое дескриптора развертывания

Мы обсуждаем XML-дескрипторы развертывания на протяжении всей этой книги. Сейчас вы, вероятно, уже достаточно знаете для того, чтобы самостоятельно писать дескрипторы развертывания. Однако все же стоит дать обзор законченного дескриптора. Далее приведены версии EJB 2.0 и 1.1 дескриптора развертывания для компонента Cabin, который мы создали в главе 4. Дескриптор развертывания компонента Cabin содержит большинство тегов, необходимых для описания объектных компонентов. Сеансовые компоненты и компоненты, управляемые сообщениями, не сильно отличаются. Различия между этими версиями небольшие, но существенные. Мы будем обращаться к этому дескриптору развертывания по ходу обсуждения в следующих разделах.

Здесь приведен дескриптор развертывания для EJB 2.0:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <description>
                Этот компонент Cabin представляет сущность «Каюта»
                прогулочного корабля.
            </description>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.cabin.CabinHomeRemote</home>
            <remote>com.titan.cabin.CabinRemote</remote>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
            <ejb-class>com.titan.cabin.CabinBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>com.titan.cabin.CabinPK</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Cabin</abstract-schema-name>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>deckLevel</field-name></cmp-field>
            <cmp-field><field-name>shipId</field-name></cmp-field>
            <cmp-field><field-name>bedCount</field-name></cmp-field>
            <primkey-field>id</primkey-field>
        </entity>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>
                Эта роль представляет всех, кому разрешен полный доступ
                к Cabin EJB.
            </description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

А это дескриптор развертывания для EJB 1.1:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <description>
                Этот компонент Cabin представляет сущность «Каюта»
                прогулочного корабля.
            </description>
            <ejb-name>CabinEJB</ejb-name>
            <home>com.titan.cabin.CabinHomeRemote</home>
            <remote>com.titan.cabin.CabinRemote</remote>
            <ejb-class>com.titan.cabin.CabinBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>name</field-name></cmp-field>
            <cmp-field><field-name>deckLevel</field-name></cmp-field>
            <cmp-field><field-name>shipId</field-name></cmp-field>
            <cmp-field><field-name>bedCount</field-name></cmp-field>
            <primkey-field>id</primkey-field>
        </entity>
    </enterprise-beans>

    <assembly-descriptor>
        <security-role>
            <description>
                Эта роль представляет всех, кому разрешен полный доступ
                к Cabin EJB.
            </description>
            <role-name>everyone</role-name>
        </security-role>

        <method-permission>
            <role-name>everyone</role-name>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>

        <container-transaction>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
```

```
        </container-transaction>
    </assembly-descriptor
</ejb-jar
```

Заголовок документа

XML-документ может начинаться с тега, указывающего на используемую версию XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Этот тег идентифицирует документ как XML-документ, который придерживается версии 1.0 спецификации XML. Указанная символьная кодировка (UTF-8) обычно поддерживается производителями EJB.

Следующий тег указывает DTD, определяющий документ. В EJB 2.0 он выглядит так:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

А в EJB 1.1 он выглядит так:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

Этот тег предоставляет URL, с которого вы (или, что более важно, инструментальные средства, обрабатывающие дескриптор развертывания) можете загрузить документ. DTD может использоваться для проверки правильности XML-документа. Это означает, что сервер EJB, развертывающий компонент, может загружать DTD и на его основании убедиться, что ваш дескриптор развертывания составлен правильно (т. е. что он правильно структурирован, в нем указаны правильные имена тегов и все теги и атрибуты имеют правильные параметры).

Этот тег также указывает имя корневого элемента документа, которым является `<ejb-jar>`. Тег `<ejb-jar>` отмечает собственно начало документа.

Тело дескриптора

Тело любого XML-документа начинается и заканчивается тегом корневого элемента документа, определенного в DTD. Для дескриптора развертывания корневой элемент называется `<ejb-jar>` и выглядит так:

```
<ejb-jar>
... остальные элементы ...
</ejb-jar>
```

Все остальные элементы должны быть вложены внутрь элемента <ejb-jar>. Внутри <ejb-jar> можно поместить следующие виды элементов:

<description> *(необязательный)*

Элемент <description> может служить для предоставления описания дескриптора развертывания. Этот элемент во многих контекстах внутри дескриптора развертывания описывает дескриптор в целом, определенные компоненты, определенные роли безопасности и т. д. В дескрипторе развертывания для компонента Cabin элемент <description> не описывает дескриптор развертывания в целом, но он предоставляет описание для самого компонента Cabin.

<display-name> *(необязательный)*

Элемент <display-name> используется инструментальными средствами (такими как мастер развертывания), работающими с дескриптором развертывания. Он предоставляет удобную визуальную метку для всего файла JAR и отдельных компонентов.

<small-icon> *и* <large-icon> *(необязательные)*

Эти элементы указывают на файлы внутри файла JAR, предоставляющие мастеру развертывания или какому-то другому средству пиктограммы (значки) для представления файла JAR. Значки должны представлять собой файлы изображений в форматах JPEG или GIF. Маленькие значки должны иметь размер 16×16 пикселов, а большие – 32×32 пиксела. Эти элементы, кроме того, используются в элементах <entity>, <session> и <message-driven> (EJB 2.0) для представления отдельных компонентов.

<enterprise-beans> *(требуется один)*

Элемент <enterprise-beans> содержит описание одного или нескольких компонентов, содержащихся в этом файле JAR. Дескриптор развертывания может содержать только один элемент <enterprise-beans>. Внутри этого элемента помещаются элементы <entity>, <session> и <message-driven> (EJB 2.0), описывающие отдельные компоненты.

<ejb-client-jar> *(необязательный)*

Элемент <ejb-client-jar> предоставляет путь к клиентскому файлу JAR, обычно содержащему все классы (включая заглушки, классы удаленных и внутренних интерфейсов и т. д.), которые потребуются клиенту для обращения к компонентам, определенным в дескрипторе развертывания. Как организуются и передаются клиенту клиентские файлы JAR, не определяется, проконсультируйтесь с документацией от производителя.

<assembly-descriptor> *(необязательный)*

Сборщик приложения или разработчик компонента добавляет к дескриптору развертывания элемент <assembly-descriptor>, указывающий, как компоненты должны применяться в реальном при-

ложении. <assembly-descriptor> содержит ряд элементов, определяющих роли безопасности, используемые для обращения к компоненту, права доступа к методам, указывающие, какие методы могут вызываться различными ролями, и атрибуты транзакции.

Эти элементы довольно просты, за исключением <enterprise-beans> и <assembly-descriptor>. Два последних содержат много другой информации. Сначала рассмотрим элемент <enterprise-beans>.

Описание компонентов

Компоненты, содержащиеся в файле JAR, описываются внутри элемента <enterprise-beans> дескриптора развертывания. До сих пор мы обсуждали только дескрипторы развертывания, содержащие единственный компонент, но в файл JAR можно упаковать несколько компонентов и описать их все внутри одного дескриптора развертывания. Например, мы можем развернуть компоненты TravelAgent, ProcessPayment, Cruise, Customer и Reservation в одном и том же файле JAR.

В EJB 2.0 кроме этого можно добавить компонент, управляемый сообщениями, такой как ReservationProcessor, разработанный нами в главе 13. Дескриптор развертывания EJB 2.0 будет выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <description>
      Этот пакет содержит все компоненты, необходимые для выполнения заказа:
      TravelAgent, ProcessPayment, Reservation, Customer, Cruise, and Cabin.
    </description>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            ...
        </session>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <remote>com.titan.customer.CustomerRemote</remote>
            ...
        </entity>
        <session>
            <ejb-name>ProcessPaymentEJB</ejb-name>
            <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
            ...
        </session>
        <message-driven>
            <ejb-name>ReservationProcessorEJB</ejb-name>
```

```
            ...
        </message-driven>
        ...
    </enterprise-beans>
    <relationships>
        ...
    </relationships>
    <assembly-descriptor>
        ...
    </assembly-descriptor>
    ...
</ejb-jar>
```

В EJB 1.1 дескриптор развертывания будет выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
    <description>
       Этот пакет содержит все компоненты, необходимые для выполнения заказа:
       TravelAgent, ProcessPayment, Reservation, Customer, Cruise, and Cabin.
    </description>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            ...
        </session>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <remote>com.titan.customer.CustomerRemote</remote>
            ...
        </entity>
        <session>
            <ejb-name>ProcessPaymentEJB</ejb-name>
            <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
            ...
        </session>
        ...
    </enterprise-beans>
    <assembly-descriptor>
        ...
    </assembly-descriptor>
    ...
</ejb-jar>
```

В этом дескрипторе элемент `<enterprise-beans>` **содержит следующие элементы, которые описывают компоненты: два элемента** `<session>`**, один элемент** `<entity>` **и (для EJB 2.0) элемент** `<message-driven>`**. Другие элементы внутри элементов** `<entity>`**,** `<session>` **и** `<message-driven>` **дают**

подробную информацию об этих компонентах. Как видите, элемент <ejb-name> задает имя компонента. Ниже мы рассмотрим все элементы, которые могут входить в описание компонента.

Преимущество развертывания сразу нескольких компонентов в том, что оно обеспечивает возможность совместного использования информации по сборке, достигаемую за счет элемента <assembly-descriptor>, расположенного за элементом <enterprise-beans>. Другими словами, компоненты могут совместно использовать безопасность и объявления, относящиеся к транзакциям, упрощая их согласованное развертывание. Например, развертывание проще выполнить, если доступом ко всем компонентам управляют одни и те же логические роли безопасности, а гарантировать, что эти роли определены согласованно, гораздо проще, если они определяются в одном месте. Кроме этого, данная стратегия упрощает согласованное применение атрибутов транзакции ко всем компонентам, т. к. можно объявить их все сразу.

Сеансовые компоненты и объектные компоненты

Элементы <session> и <entity>, описывающие сеансовые и объектные компоненты, как правило, содержат много вложенных элементов. Наборы разрешенных для них подэлементов очень похожи, поэтому мы будем рассматривать элементы <session> и <entity> совместно.

Как и элемент <ejb-jar>, элементы <session> и <entity> могут содержать необязательные элементы <description>, <display-name>, <small-icon> и <large-icon>. Их смысл достаточно очевиден и в обоих случаях такой же, как и в элементе <ejb-jar>. <description> позволяет снабдить компонент описательным комментарием. <display-name> используется инструментальными средствами развертывания для представления компонента. А два элемента с изображениями служат для представления компонента в графических средах. Элементы изображений должны указывать на изображения в форматах JPEG или GIF, находящиеся внутри файла JAR. Другие элементы более интересны:

<ejb-name> *(требуется один)*

Задает имя компонента. Он включается в элемент <methodx> для указания границ объявлений методов для корректного компонента. На протяжении всей книги в качестве значений для <ejb-name>-компонента применяются имена в форме «NameEJB». В других распространенных соглашениях имена задаются в форме «NameBean» и «TheName».

<home> *(EJB 1.1: требуется один, EJB 2.0: необязательный)*

Указывает полное имя класса удаленного внутреннего интерфейса компонента.

<remote> *(EJB 1.1: требуется один, EJB 2.0: необязательный)*

Указывает полное имя класса удаленного интерфейса компонента.

`<local-home>` *(EJB 2.0: необязательный)*

Указывает полное имя класса локального внутреннего интерфейса компонента.

`<local>` *(EJB 2.0: необязательный)*

Указывает полное имя класса локального интерфейса компонента.

`<ejb-class>` *(требуется один)*

Указывает полное имя класса компонента.

`<primkey-field>` *(необязательный, только для объектных компонентов)*

Определяет поле первичного ключа для объектных компонентов, использующих постоянство, управляемое контейнером. Значение этого элемента представляет собой имя поля, выступающего в качестве первичного ключа. Оно не используется в случаях, когда компонент имеет составной первичный ключ или когда объектный компонент управляет своим постоянством самостоятельно. В компоненте Cabin элемент `<primkey-field>` содержит поле CMP `id`. Этот элемент рассмотрен более подробно в разделе «Определение первичных ключей» далее в этой главе.

`<prim-key-class>` *(требуется один, только для объектных компонентов)*

Определяет класс первичного ключа для объектных компонентов и содержит полное имя класса первичного ключа. Не имеет никакого значения, определяете ли вы пользовательский составной первичный ключ или простой `<primkey-field>`, такой как `Integer`, `String`, `Date` и т. д. Откладывая определение класса первичного ключа до времени развертывания, задайте его тип в этом элементе как `java.lang.Object`.

`<persistence-type>` *(требуется один, только для объектных компонентов)*

Указывает, что объектный компонент использует либо постоянство, управляемое контейнером, либо постоянство, реализуемое компонентом. Этот элемент может иметь одно из двух значений: `Container` или `Bean`.

`<reentrant>` *(требуется один, только для объектных компонентов)*

Содержит информацию о том, разрешает ли компонент циклические вызовы (повторные вхождения). Может иметь одно из двух значений: `true` или `false`. Первое означает, что компонент допускает повторные вхождения, а второе, – что при повторном вхождении компонент будет генерировать исключение.

`<cmp-version>` *(EJB 2.0: необязательный)*

Описывает версию постоянства, управляемого контейнером, под которое развертывается объектный компонент. Для обратной совместимости контейнеры EJB должны поддерживать и EJB 2.0 CMP, и EJB 1.1 CMP. Этот элемент может содержать одно из двух значений: 2.x – для EJB 2.0 или 1.x – для EJB 1.1.

<abstract-schema-name> *(EJB 2.0: необязательный)*

Уникально идентифицирует объектные компоненты в файле JAR так, чтобы на них можно было ссылаться из операторов EJB QL. Этот метод более подробно описан далее в разделе «Объявление элементов EJB QL».

<cmp-field> *(несколько или ни одного, только для объектных компонентов)*

Применяется в объектных компонентах с постоянством, управляемым контейнером. Элемент <cmp-field> должен существовать для каждого управляемого контейнером поля класса компонента. Каждый элемент <cmp-field> может содержать элемент <description> и должен включать элемент <field-name>. <description> представляет собою необязательный комментарий, описывающий поле. <field-name> должен присутствовать и содержать имя одного из полей CMP компонента. В EJB 2.0 он должен соответствовать имени абстрактного метода доступа (например, deckLevel для getDeckLevel()/setDeckLevel()). В EJB 1.1 <cmp-field> должен соответствовать имени одного из полей, объявленных в классе компонента. Контейнер будет управлять постоянством данного поля CMP. В следующем фрагменте дескриптора показано несколько объявлений <cmp-field> для компонента Cabin:

```
<cmp-field>
    <description>Это первичный ключ </description>
    <field-name>id</field-name>
</cmp-field>
<cmp-field>
    <field-name>name</field-name>
</cmp-field>
<cmp-field>
    <field-name>deckLevel</field-name>
</cmp-field>
<cmp-field>
    <field-name>shipId</field-name>
</cmp-field>
<cmp-field>
    <field-name>bedCount</field-name>
</cmp-field>
```

<env-entry> *(несколько или ни одного)*

Объявляет элемент окружения, доступный через JNDI ENC. Использование элементов окружения в компоненте и дескрипторе развертывания рассмотрено далее в разделе «Элементы окружения».

<ejb-ref> *(несколько или ни одного)*

Объявляет удаленную ссылку компонента, которая является доступной через JNDI ENC. Механизм обеспечения доступа к ссыл-

кам через ENC описан более подробно далее в разделе «Ссылки на другие компоненты».

`<ejb-local-ref>` *(EJB 2.0: несколько или ни одного)*

Объявляет локальную ссылку компонента, доступную через JNDI ENC. Механизм обеспечения доступа к ссылкам через ENC описан более подробно далее в разделе «Ссылки на другие компоненты».

`<resource-ref>` *(несколько или ни одного)*

Объявляет ссылку на фабрику подключений, доступную через JNDI ENC. Примером фабрики ресурсов является `javax.sql.DataSource`, применяемый для получения подключений к базам данных. Этот элемент подробно рассматривается в разделе «Ссылки на внешние ресурсы» далее в данной главе.

`<resource-env-ref>` *(EJB 2.0: несколько или ни одного)*

Описывает дополнительные «управляемые объекты», необходимые ресурсу. Элемент `<resource-env-ref>` и управляемые объекты обсуждаются более подробно в разделе «Ссылки на внешние ресурсы» далее в этой главе.

`<security-role-ref>` *(несколько или ни одного)*

Предназначен для указания в дескрипторе развертывания ролей безопасности и сопоставления их с ролями безопасности реального окружения компонента времени выполнения. Этот метод более подробно описан далее в разделе «Роли безопасности».

`<security-identity>` *(EJB 2.0: необязательный)*

Определяет `Principal`, под которым будет выполняться метод. Этот элемент описан более подробно в последующем разделе «Определение ролей безопасности и разрешений методов».

`<session-type>` *(требуется один, только для сеансовых компонентов)*

Указывает, является ли сеансовый компонент компонентом без состояния или с состоянием. Данный элемент может иметь одно из двух значений: `Stateful` или `Stateless`.

`<transaction-type>` *(требуется один, только для сеансовых компонентов)*

Указывает, управляет ли сеансовый компонент своими собственными транзакциями сам, или его транзакциями управляет контейнер. Данный элемент может иметь одно из двух значений: `Bean` или `Container`. Компонент, который сам управляет своими собственными транзакциями, не будет содержать объявлений, связанных с контейнерными транзакциями, в разделе `<assembly-descriptor>` дескриптора развертывания.

`<query>` *(EJB 2.0: несколько или ни одного)*

Содержит оператор EJB QL, связанный с поисковым методом или методом выборки. Оператор EJB QL определяет, как поисковый ме-

тод или метод выборки должны функционировать во время выполнения. Этот элемент описан более подробно далее в разделе «Объявление элементов EJB QL».

Компоненты, управляемые сообщениями

Элемент <message-driven> описывает операции развертывания компонента, управляемого сообщениями. Элементы <message-driven> расположены внутри элемента <enterprise-beans> после элементов <entity> и <session>.

Как и элементы <entity> и <session>, элемент <message-driven> может содержать необязательные элементы <description>, <display-name>, <small-icon> и <large-icon>. Эти элементы используются, в первую очередь, визуальными инструментальными средствами развертывания для представления компонента, управляемого сообщениями. Также элемент <message-driven> требует объявления элементов <ejb-name>, <ejb-class>, <transaction-type> и <security-identity>. Кроме того, он содержит стандартные JNDI-элементы ENC <env-entry>, <ejb-ref>, <ejb-local-ref>, <resource-ref> и <resource-env-ref>. Они достаточно очевидны и означают то же самое, что и в элементах <entity> и <session>.

Элементы, специфичные для компонента, управляемого сообщениями:

<message-selector>

Селекторы сообщений позволяют MDB быть более избирательным относительно сообщений, которые он получает из заданной темы или очереди. Селекторы сообщений используют свойства Message в качестве аргументов в условных выражениях.[1] Эти условные выражения используют булеву логику для определения, какие сообщения должны быть направлены клиенту. Синтаксис селекторов сообщений может вызывать проблемы при разборе XML. См. врезку, озаглавленную «Разделы CDATA»

<acknowledge-mode>

Данный элемент учитывается контейнером только тогда, когда компонент, управляемый сообщениями, использует управляемые компонентом транзакции. При использовании транзакций, управляемых контейнером, он игнорируется. Он определяет, какой тип подтверждения будет использоваться. Его значение может быть либо Auto-acknowledge, либо Dups-ok-acknowledge. Первое подтверждает сообщения немедленно, второе может откладывать подтверждения, ради повышения производительности, но может привести и к двойным экземплярам, и к повторной посылке сообщений.

[1] Селекторы сообщений также могут использовать информацию из заголовков сообщений, но это выходит за рамки данной книги.

<message-driven-destination>

Этот элемент определяет тип пункта назначения, на который MDB подписывается или который прослушивает. Возможные значения для этого элемента – javax.jms.Queue и javax.jms.Topic.

Определение первичных ключей

Если в компоненте имеется отдельное поле, которое может служить естественным уникальным идентификатором, то это поле может выступать в качестве первичного ключа. Кроме того, в качестве составного первичного ключа может применяться пользовательский первичный ключ. В компоненте Cabin, например, тип первичного ключа мог бы быть CabinPK, который был бы связан с полями id и name класса компонента, как показано здесь (для большей наглядности CabinBean использует постоянство, реализуемое компонентом):

```
public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;
    public int ship;
    public int bedCount;

    public CabinPK ejbCreate(int id, String name) {
        this.id = id;
        this.name = name;
        return null;
    }
    ...
}
```

В главе 4 вместо пользовательского класса CabinPK мы применили подходящую примитивную обертку java.lang.Integer и определили CabinBean как:

```
public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;
    public int ship;
    public int bedCount;

    public Integer ejbCreate(int id) {
        this.id = id;
        return null;
    }
    ...
}
```

Разделы CDATA

В элементы <message-selector>, используемые компонентами, управляемыми сообщениями, и элементами <ejb-ql>, часто приходится включать символы, имеющие специальное значение в XML, например символы < и >. Данные символы могут вызвать ошибки при синтаксическом разборе, если не использовать разделы CDATA.

Раздел CDATA имеет вид <![CDATA[литерный-текст]]>. Встретив раздел CDATA, процессор XML не пытается анализировать содержание, заключенное в разделе CDATA.

Ниже показан раздел CDATA в элементе <message-selector>:

```
<message-selector>
<![CDATA[
    TotalCharge >500.00 AND ((TotalCharge /ItemCount)>=75.00)
    AND State IN ('MN','WI','MI','OH')";
]]>
</message-selector>
```

Раздел CDATA в элементе <ejb-ql>:

```
<query>
    <query-method>
        ...
    </query-method>
    <ejb-ql>
        <![CDATA[
           SELECT OBJECT( r ) FROM Reservation r
           WHERE r.amountPaid  > 300.00
        ]]>
    </ejb-ql>
```

Это сильно упрощает дело. Вместо того чтобы тратить время на определение пользовательского первичного ключа, такого как CabinPK, мы просто выбираем подходящую обертку. С этой целью нам необходимо добавить в дескриптор развертывания для компонента Cabin элемент <primkey-field>, чтобы он знал, какое поле будет выступать в качестве первичного ключа. Нам также потребуется изменить элемент <primkey-class>, чтобы указать, что для представления первичного ключа применяется класс Integer. В следующем коде показано, как следует изменить дескриптор развертывания для компонента Cabin для того, чтобы он использовал тип Integer в качестве поля первичного ключа:

```
<entity>
    <description>
     Этот компонент Cabin представляет сущность «Каюта» прогулочного корабля.
    </description>
    <ejb-name>CabinEJB</ejb-name>
```

```
    <home>com.titan.cabin.CabinHome</home>
    <remote>com.titan.cabin.Cabin</remote>
    <ejb-class>com.titan.cabin.CabinBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>

    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>deckLevel</field-name></cmp-field>
    <cmp-field><field-name>ship</field-name></cmp-field>
    <cmp-field><field-name>bedCount</field-name></cmp-field>
    <primkey-field>id</primkey-field>
</entity>
```

Типы полей простых первичных ключей не ограничены примитивными классами обертки (`Byte`, `Boolean`, `Integer` и т. д.). В качестве первичного ключа может выступать любое управляемое контейнером поле, если оно является сериализуемым. Типы `String`, вероятно, наиболее распространены, но также могут применяться и другие типы, такие как `java.lang.StringBuffer`, `java.util.Date` или даже `java.util.Hashtable`. Первичными ключами могут быть также и пользовательские типы, при условии, что они сериализуемые. Конечно, при выборе первичного ключа нужно исходить из здравого смысла: поскольку он применяется в качестве индекса для данных в базе данных, он должен быть достаточно легким.

Отложенное определение первичного ключа

В постоянстве, управляемом контейнером, разработчику компонента разрешается отложить определение первичного ключа, оставив его определение управляющему развертыванием компонентов. Эта возможность может потребоваться, если, например, первичный ключ генерируется базой данных, а в классе компонента нет полей, управляемых контейнером. Такой подход мог бы быть реализован в контейнерах, тесно связанных с базой данных или со старыми системами, которые автоматически генерируют первичные ключи. Этот подход также привлекателен для производителей, продающих компоненты с урезанными возможностями, поскольку это делает компонент более переносимым. В следующем коде показано, как для объектного компонента, использующего постоянство, управляемое контейнером, определение первичного ключа откладывается до времени развертывания:

```
// Класс компонента с отложенным первичным ключом
public class HypotheticalBean implements javax.ejb.EntityBean {
    ...
    public java.lang.Object ejbCreate() {
```

```
        ...
        return null;
    }
    ...
}

// Внутренний интерфейс компонента с отложенным первичным ключом
public interface HypotheticalHome extends javax.ejb.EJBHome {
    public Hypothetical create() throws ...;
    public Hypothetical findByPrimaryKey(java.lang.Object key) throws ...;
}
```

Здесь приведена относящаяся к нему часть дескриптора развертывания:

```
// Объявление поля первичного ключа для гипотетического компонента
...
<entity>
    <ejb-name>HypotheticalEJB</ejb-name>
    ...
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Object</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>creationDate</field-name></cmp-field>
    ...
</entity>
```

Первичный ключ имеет тип `java.lang.Object`, поэтому взаимодействие клиентского приложения с ключом компонента ограничено типом `Object` и его методами.

Элементы окружения

В дескрипторе развертывания можно определять элементы окружения, представляющие собой значения, похожие на свойства, которые могут читаться компонентом во время выполнения. По элементам окружения компонент может подстраивать свое поведение, выяснять, как он был развернут, и т. д.

Для определения элементов окружения предназначен элемент `<env-entry>`. Этот элемент содержит подэлементы `<description>` (необязательный), `<env-entry-name>` (обязательный), `<env-entry-type>` (обязательный) и `<env-entry-value>` (необязательный). Ниже приводится типичное объявление `<env-entry>`:

```
<env-entry>
    <env-entry-name>minCheckNumber</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>2000</env-entry-value>
</env-entry>
```

<env-entry-name> относится к контексту "java:comp/env". Например, к элементу minCheckNumber можно обратиться, указав путь "java:comp/env/minCheckNumber" в поиске через JNDI ENC:

```
InitialContext jndiContext = new InitialContext();
Integer miniumValue = (Integer)
    jndiContext.lookup("java:comp/env/minCheckNumber");
```

<env-entry-type> может иметь тип String или один из нескольких примитивных типов-оберток, включающих Integer, Long, Double, Float, Byte, Boolean и Short.

<env-entry-value> не относится к обязательным. Его значение может быть определено разработчиком компонента или оставлено сборщику приложения или управляющему развертыванием.

Для того чтобы сделать элемент доступным для метода EJBContext.getEnvironment() может быть использован подконтекст "java:comp/env/ejb10-properties". Было объявлено, что данная особенность не рекомендована, но с ее помощью можно развернуть компоненты EJB 1.0 внутри сервера EJB 1.1. Для элементов этого подконтекста <ejb-entry-type> всегда должен быть установлен в java.lang.String. Приведем пример:

```
<env-entry>
    <description>
        Это свойство доступно через EJBContext.getEnvironment()
    </description>
    <env-entry-name>ejb10-properties/minCheckNumber</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-name>
    <env-entry-value>20000</env-entry-value>
</env-entry>
```

Ссылки на другие компоненты

В EJB 2.0 ссылки на другие компоненты могут быть либо локальными, либо удаленными. В EJB 1.1 ссылки на другие компоненты всегда представляют собой удаленные ссылки.

Удаленные ссылки

Элемент <env-ref> предназначен для определения ссылок на другие компоненты, расположенные в пределах JNDI ENC. Это сильно облегчает компонентам обращение к другим компонентам. Они могут использовать JNDI для поиска ссылки на внутренний интерфейс любых компонентов, которые им необходимы.

Элемент <env-ref> содержит подэлементы <description> (необязательный), <ejb-ref-name> (обязательный), <ejb-ref-type> (обязательный), <remote> (обязательный), <home> (обязательный) и <ejb-link> (необязательный). Приведем типичное объявление <env-ref>:

```
<ejb-ref>
    <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
    <remote>com.titan.processpayment.ProcessPaymentHomeRemote</remote>
</ejb-ref>
```

<ejb-ref-name> относится к контексту "java:comp/env". Рекомендуется, но не требуется, чтобы его имя было помещено в подконтекст ejb/. В соответствии с этим соглашением обращение к внутреннему объекту компонента ProcessPayment будет происходить по пути "java:comp/env/ejb/ProcessPaymentHomeRemote". В следующем коде показано, как клиентский компонент организует поиск ссылки на компонент ProcessPayment по этому контексту:

```
InitialContext jndiContext = new InititalContext();
Object ref = jndiContext.lookup("java:comp/env/ejb/
ProcessPaymentHomeRemote");
ProcessPaymentHomeRemote home = (ProcessPaymentHomeRemote)
    PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);
```

<ejb-ref-type> может иметь одно из двух значений, Entity или Session, в зависимости от того, является ли компонент объектным или сеансовым.

Элемент <home> определяет полное имя класса внутреннего интерфейса компонента. Элемент <remote> определяет полное имя класса удаленного интерфейса компонента.

Если компонент, на который ссылается элемент <ejb-ref>, развернут в том же самом дескрипторе развертывания (определен внутри того же элемента <ejb-jar>), то элемент <ejb-ref> может быть связан с объявлением компонента с помощью элемента <ejb-link>. Если, например, компонент TravelAgent ссылается на компонент ProcessPayment, объявленный в том же дескрипторе развертывания, то элемент <ejb-ref> для компонента TravelAgent может с помощью элемента <ejb-link> отображать свои элементы <ejb-ref> на компонент ProcessPayment. Значение <ejb-link> должно соответствовать одному из значений <ejb-name>, объявленных в этом же дескрипторе развертывания. Далее приведена часть дескриптора развертывания, в котором используется элемент <ejb-link>:

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>TravelAgentEJB</ejb-name>
            <remote>com.titan.travelagent.TravelAgentRemote</remote>
            ...
            <ejb-ref>
                <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
```

```
            <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
            <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
             <ejb-link>ProcessPaymentEJB</ejb-link>
          </ejb-ref>
          ...
      </session>

      <session>
          <ejb-name>ProcessPaymentEJB</ejb-name>
          <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
          ...
      </session>
      ...
    </enterprise-beans>
    ...
</ejb-jar>
```

Разработчики на EJB 2.0, возможно, только выиграют от применения элемента <ejb-local-ref> для получения ссылок на компоненты, расположенные в том же файле JAR, если только компонент, на который необходимо сослаться, будет иметь набор локальных компонентных интерфейсов. В этом случае для получения удаленной ссылки на компонент необходимо использовать элементы <ejb-link> и <ejb-ref>.

EJB 2.0: Локальные ссылки

Дескриптор развертывания также предоставляет специальный набор тегов (элементы <ejb-local-ref>) для объявления локальных компонентных ссылок, т. е. ссылок на компоненты, размещенные в том же контейнере и развернутые в том же файле EJB JAR. Элементы <ejb-local-ref> объявляются сразу после элементов <ejb-ref>:

```
<ejb-local-ref>
    <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
    <local>com.titan.cruise.CruiseLocal</local>
    <ejb-link>CruiseEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
    <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>com.titan.cabin.CabinHomeLocal</local-home>
    <local>com.titan.cabin.CabinLocal</local>
    <ejb-link>CabinEJB</ejb-link>
</ejb-local-ref>
```

Элемент <ejb-local-ref> определяет имя компонента в рамках ENC, объявляет тип компонента и дает имена его локальным компонентным интерфейсам. Эти элементы должны быть явно связаны с другими совмещенными компонентами с помощью элемента <ejb-link>, но

это не обязательно: сборщик приложения или управляющий развертыванием могут сделать это позднее. Значение элемента <ejb-link>, вложенного в <ejb-local-ref>, должно совпадать с <ejb-name> необходимого компонента из того же файла JAR.

Во время развертывания инструментальные средства контейнера EJB связывают локальные ссылки, объявленные в элементах <ejb-local-ref>, с объектными компонентами, размещенными в той же контейнерной системе.

Компоненты, объявляемые в элементах <ejb-local-ref>, представляют собой локальные компоненты и поэтому не требуют применения метода PortableRemoteObject.narrow() для сужения ссылки. Это можно сделать посредством простой встроенной операции приведения типов:

```
InitialContext jndiContext = new InititalContext();
CabinHome home = (CabinHome)
    jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");
```

Ссылки на внешние ресурсы

JNDI ENC применяется в компонентах также и для поиска внешних ресурсов, таких как соединения с базами данных, к которым им нужно получить доступ. Делается это при помощи механизма, похожего на механизм, применяемый для ссылки на другие компоненты и элементы окружения: внешние ресурсы сопоставляются с именами в рамках пространства имен JNDI ENC. Для внешних ресурсов сопоставление выполняется с помощью элемента <resource-ref>.

Элемент <resource-ref> содержит подэлементы <description> (необязательный), <res-ref-name> (обязательный), <res-type> (обязательный) и <res-auth> (обязательный).

Приведем объявление <resource-ref>, используемое для фабрики подключений DataSource:

```
<resource-ref>
    <description>DataSource для базы данных «Титан»</description>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

<res-ref-name> относится к контексту "java:comp/env". Хотя и не обязательно, но желательно размещать фабрики подключений внутри подконтекста, описывающего тип ресурса. Например:

- jdbc/ для фабрики JDBC DataSource
- jms/ для фабрик JMS QueueConnectionFactory и TopicConnectionFactory
- mail/ для фабрики сессий JavaMail
- url/ для фабрики javax.net.URL

Далее показано, как компонент должен использовать JNDI для поиска ресурса (в данном случае – DataSource):

```
InitialContext jndiContext = new InitialContext();
DataSource source = (DataSource)
    jndiContext.lookup("java:comp/env/jdbc/titanDB");
```

Элемент <res-type> предназначен для описания полного имени класса фабрики подключений. В этом примере <res-type> содержит javax.sql.DataSource.

Элемент <res-auth> сообщает серверу, кто отвечает за аутентификацию. Он может иметь одно из двух значений: Container или Application. Если задан Container, то аутентификация (вход в систему) для использования ресурса будет автоматически выполняться контейнером, как будет указано во время развертывания. Если же элемент <res-auth> имеет значение Application, то аутентификация перед использованием ресурса должна выполняться самим компонентом. В следующем коде показано, как компонент может подписаться на фабрику подключений, если для <res-auth> определено значение Application:

```
InitialContext jndiContext = new InitialContext();
DataSource source = (DataSource)
    jndiContext.lookup("java:comp/env/jdbc/titanDB");

String loginName = ejbContext.getCallerPrincipal().getName();
String password = ...; // получаем пароль из некоторого источника

// используем имя и пароль для установки связи с базой данных
java.sql.Connection con = source.getConnection(loginName, password);
```

EJB 2.0: Дополнительные управляемые объекты

В дополнение к фабрике ресурсов, описываемой элементом <resource-ref>, некоторые ресурсы могут иметь другие управляемые объекты, которые должны быть получены из JNDI ENC. *Управляемый объект* (*administered object*) – это ресурс, конфигурируемый во время развертывания и управляемый контейнером EJB во время выполнения. Например, для того чтобы использовать JMS, разработчик компонента должен получить и объект-фабрику JMS, и объект пункта назначения:

```
TopicConnectionFactory factory = (TopicConnectionFactory)
    jndiContext.lookup("java:comp/env/jms/TopicFactory");

Topic topic = (Topic)
    jndiContext.lookup("java:comp/env/ejb/TicketTopic");
```

И фабрика JMS, и пункт назначения представляют собой управляемые объекты, которые должны быть получены из JNDI ENC. Элемент <resource-ref> применяется для объявления фабрики JMS, а элемент <resource-env-ref> – для объявления пункта назначения:

```
<resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>
    <res-type>javax.jms.TopicConnectionFactory</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
<resource-env-ref>
    <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

Во время развертывания выполняется сопоставление JMS `TopicConnectionFactory` или `QueueConnectionFactory` и объектов `Topic` или `Queue`, объявленных в элементах `<resource-ref>` и `<resource-env-ref>` с фабрикой и темой JMS.

EJB 2.0: Разделяемые ресурсы

Когда несколько компонентов в единице работы (транзакции) одновременно обращаются к одному и тому же ресурсу, необходимо сконфигурировать сервер EJB для совместного использования (разделения) этого ресурса. Разделение (sharing) ресурса означает, что каждый компонент обращается к ресурсу посредством одного и того же подключения (например, к базе данных или провайдеру JMS), что более эффективно, чем организация отдельных подключений к ресурсу.

Например, в компоненте TravelAgent метод `bookPassage()` использует компоненты ProcessPayment и Reservation для регистрации пассажира на круиз. В случае если оба эти компонента обращаются к одной базе данных, они должны с целью повышения эффективности совместно использовать свои подключения к ресурсу. Контейнеры Enterprise JavaBeans работают в режиме разделения ресурсов по умолчанию, но эту установку можно отключить или включить явно с помощью элемента `<resource-ref>`:

```
<resource-ref>
    <res-ref-name>jdbc/titanDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Элемент `<res-sharing-scope>` не относится к обязательным и может содержать либо `Shareable`, указывая, что подключения должны использоваться совместно локальными транзакциями, либо `Unshareable`, говоря об обратном. Если ничего не указано, принимается значение по умолчанию – `Shareable`.

Продвинутые разработчики могут столкнуться с ситуацией, когда совместное использование ресурсов нежелательно. Это случается относительно редко, но возможность отключить совместное использование ресурсов пригодится в этом случае. При отсутствии веских причин для отключения совместного использования ресурсов я бы не рекомендовал это делать.

Роли безопасности

Элемент `<security-role-ref>` предназначен для задания ролей безопасности, используемых компонентом, и сопоставления их с ролями безопасности, действующими для среды времени выполнения. Он может содержать три подэлемента: необязательный `<description>`, обязательный `<role-name>` и необязательный `<role-link>`.

Далее показано, как определяются роли безопасности. Имя роли, указанное в методе `EJBContext.isCallerInRole(String roleName)`, должно быть статически определенным (оно не может наследоваться во время выполнения) и должно быть объявлено в дескрипторе развертывания с помощью элемента `<security-role-ref>`:

```
<-- определение security-role-ref для компонента Account -->
<entity>
    <ejb-name>AccountEJB</ejb-name>
    ...
    <security-role-ref>
        <description>
            Тот, кто обращается к этому компоненту, должен быть соотнесен
            с этой ролью в случае, если сумма платежа больше $10,000
        </description>
        <role-name>Manager</role-name>
        <role-link>Administrator</role-link>
    </security-role-ref>
    ...
</entity>
```

Элемент `<role-name>`, определенный в дескрипторе развертывания, должен быть соотнесен с именем роли, указанным в методе `EJBContext.isCallerInRole()`. Здесь показано, как имя роли используется в коде компонента:

```
// Компонент Account вызывает метод isCallerInRole()
public class AccountBean implements EntityBean {
    int id;
    double balance;
    EntityContext context;

    public void withdraw(Double withdraw) throws AccessDeniedException {

        if (withdraw.doubleValue() > 10000) {
            boolean isManager = context.isCallerInRole("Manager");
```

```
            if (!isManager) {
                // Выдавать суммы более 10000 могут только менеджеры
                throw new AccessDeniedException();
            }
        }
        balance = balance - withdraw.doubleValue();

    }
    ...
}
```

Элемент <role-link> не обязателен и может применяться для сопоставления имени роли, фигурирующем в компоненте, с логической ролью, определенной в элементе <security-role> раздела <assembly-descriptor> дескриптора развертывания. Если не указано никакого элемента <role-link>, то во время развертывания необходимо сопоставить <security-role-ref> существующей роли безопасности целевого окружения.

Объявление элементов EJB QL

Операторы EJB QL объявляются в элементах <query> дескриптора развертывания объектного компонента. В следующем листинге можно увидеть, что в элементах <query> дескриптора развертывания для компонента Cruise были объявлены методы findByName() и ejbSelectShips():

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>ShipEJB</ejb-name>
            ...
            <abstract-schema-name>Ship</abstract-schema-name>
            ...
        </entity>
        <entity>
            <ejb-name>CruiseEJB</ejb-name>
            ...
            <reentrant>False</reentrant>
            <abstract-schema-name>Cruise</abstract-schema-name>
            <cmp-version>2.x</cmp-version>
            <cmp-field>
                <field-name>name</field-name>
            </cmp-field>
            <primkey-field>id</primkey-field>
            <query>
                <query-method>
                    <method-name>findByName</method-name>
                    <method-params>
                        <mehod-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <ejb-ql>
```

```
                SELECT OBJECT(c) FROM Cruise c WHERE c.name = ?1
            </ejb-ql>
        </query>
        <query>
            <query-method>
                <method-name>ejbSelectShips</method-name>
                <method-params></method-params>
            </query-method>
            <result-type-mapping>Remote</result-type-mapping>
            <ejb-ql>
                SELECT OBJECT(s) FROM Ship AS s
            </ejb-ql>
        </query>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Элемент <query> содержит два основных элемента. Элемент <query-method> идентифицирует поисковый метод удаленного и/или локального внутреннего интерфейса, а элемент <ejb-ql> объявляет оператор EJB QL. Элемент <query> связывает оператор EJB QL с требуемым поисковым методом. Синтаксис, принятый в EJB QL, может вызвать сбои в работе синтаксического анализатора XML. Для более полной информации см. врезку «Разделы CDATA» выше.

Если два поисковых метода локального и удаленного внутреннего интерфейса имеют одинаковые имена и параметры, то объявление запроса применяется к обоим методам. Контейнер возвратит правильный тип для каждого метода запроса: удаленный внутренний интерфейс возвратит одни или несколько удаленных компонентных объектов, а локальный внутренний – один или несколько локальных компонентных объектов. Это дает возможность задавать поведение и локальных, и удаленных внутренних поисковых методов с помощью одного элемента <query>, что удобно, если необходимо, чтобы локальные клиенты имели доступ к тем же поисковым методам, что и удаленные клиенты.

В элементе <result-type-mapping> можно указать, какие компонентные объекты, локальные или удаленные, должен возвращать метод выборки. Значение Local указывает, что метод должен возвращать локальные компонентные объекты, Remote означает удаленные компонентные объекты. Если элемент <result-type-mapping> отсутствует, принимается значение по умолчанию – Local. В элементе <query>, соответствующем методу ejbSelectShips(), значение <result-type-mapping> объявлено как Remote. Это означает, что запрос должен возвращать удаленные типы компонентного объекта (т. е. удаленные ссылки на компонент Ship).

Все объектные компоненты, на которые ссылается оператор EJB QL, должны иметь специальный определитель, называемый *именем абстрактной схемы* (*abstract schema name*), который задается в эле-

менте <abstract-schema-name>. Элементы <abstract-schema-name> должны иметь уникальные имена. Никакие два объектных компонента не могут иметь совпадающие имена абстрактных схем. В элементе, описывающем компонент Cruise, имя абстрактной схемы объявлено как Cruise, а имя абстрактной схемы компонента Ship – Ship. Элемент <ejb-ql> содержит оператор EJB QL, который использует этот идентификатор в своей секции FROM.

Из главы 7 нам известно, что абстрактная схема постоянства объектного компонента задается его элементами <cmp-field> и <cmr-field>. Абстрактное имя схемы также представляет собой важную часть абстрактной схемы постоянства. Операторы EJB QL всегда выражаются в терминах абстрактных схем постоянства объектных компонентов. В EJB QL абстрактные имена схем служат для идентификации типов объектных компонентов; поля постоянства, управляемого контейнером (CMP), – для идентификации данных определенных объектных компонентов; и поля отношений, управляемые контейнером (CMR), – для создания путей перемещения от одного объектного компонента к другому.

EJB 2.0: Описание отношений

Классы объектных компонентов CMP 2.0 определяются посредством абстрактных методов доступа, которые представляют виртуальные поля постоянства и отношений. Как мы узнали из глав 6, 7 и 8, сами действительные поля не объявляются в классах объектных компонентов. Но характеристики этих полей подробно описываются в XML-дескрипторе развертывания, используемом этим компонентом. *Абстрактная схема постоянства* (*abstract persistence schema*) представляет собой набор XML-элементов в дескрипторе развертывания, описывающих поля постоянства и отношений. Абстрактная программная модель (т. е. с абстрактными методами доступа) и определенная помощь со стороны управляющего развертыванием дают контейнерному средству достаточно информации для того, чтобы связать компонент и его отношения с другими объектными компонентами.

Отношения между объектными компонентами описываются в разделе <relationships> XML-дескриптора развертывания. Раздел <relationships> находится между разделами <enterprise-beans> и <assembly-descriptor>. Внутри элемента <relationships> каждое отношение типа «объект–объект» определяется в отдельном элементе <ejb-relation>:

```
<ejb-jar>
    <enterprise-beans>
    ...
    </enterprise-beans>
    <relationships>
```

```
        <ejb-relation>
        ...
        </ejb-relation>
        <ejb-relation>
        ...
        </ejb-relation>
    </relationships>
    <assembly-descriptor>
    ...
    </assembly-descriptor>
</ejb-jar>
```

Определение полей отношения требует внесения в XML-дескриптор развертывания элемента <ejb-relation> для каждого отношения «объект–объект». Эти элементы <ejb-relation> дополняют абстрактную программную модель. Для каждой пары абстрактных методов доступа, определяющей поле отношения, в дескрипторе развертывания существует элемент <ejb-relation>. EJB 2.0 требует, чтобы объектные компоненты, участвующие в каком-либо отношении, были определены в том же XML-дескрипторе развертывания.

Приведем фрагмент листинга дескриптора развертывания для компонентов Customer и Address, в котором выделены элементы, определяющие отношения:

```
<ejb-jar>
    ...
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <local-home>com.titan.customer.CusomterLocalHome</local-home>
            <local>com.titan.customer.CustomerLocal</local>
            ...
        </entity>
        <entity>
            <ejb-name>AddressEJB</ejb-name>
            <local-home>com.titan.address.AddressLocalHome</local-home>
            <local>com.titan.address.AddressLocal</local>
            ...
        </entity>
        ...
    </enterprise-beans>

    <relationships>
        <ejb-relation>
            <ejb-relation-name>Customer-Address</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    Customer-has-an-Address
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
```

```
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>homeAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Address-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</ejb-jar>
```

Все отношения, существующие между компонентом Customer и другими объектными компонентами, такими как CreditCard, Address и Phone, требуют, чтобы в дополнение к абстрактным методам доступа мы определили элемент `<ejb-relation>`.

Каждое отношение может иметь имя, объявляемое в элементе `<ejb-relation-name>`. Это имя идентифицирует отношение для тех, кто читает дескриптор развертывания, и для инструментальных средств развертывания, но оно не обязательно.

Каждый элемент `<ejb-relation>` содержит ровно два элемента `<ejb-relationship-role>`, по одному для каждого участника отношения. В предыдущем примере первый элемент `<ejb-relationship-role>` определял роль компонента Customer в отношении. Мы знаем это, потому что в элементе `<relationship-role-source>` значение `<ejb-name>` определено как `CustomerEJB`. А `CustomerEJB` указан в элементе `<ejb-name>`, используемом при первоначальном объявлении компонента Customer в разделе `<enterprise-beans>`. Подэлемент `<ejb-name>` элемента `<relationship-role-source>` должен всегда соответствовать элементу `<ejb-name>` в разделе `<enterprise-beans>`.

Элемент `<ejb-relationship-role>` также объявляет размерность, или *множественность* (*multiplicity*), роли. Элемент `<multiplicity>` может содержать или `One`, или `Many`. В нашем случае элемент `<multiplicity>` для компонента Customer имеет значение `One`, а это означает, что каждый компонент Address имеет отношение ровно с одним компонентом Customer. В элементе `<multiplicity>` для компонента Address также определено `One`, т. е. каждый компонент Customer имеет отношение ровно с одним компонентом Address. Если бы компонент Customer имел отношение с многими компонентами Address, то значение элемента `<multiplicity>` для компонента Address было бы установлено в `Many`.

Если компонент, описываемый элементом <ejb-relationship-role>, содержит ссылку на другой компонент в отношении, то данная ссылка должна быть объявлена в элементе <cmr-field> как управляемое контейнером поле отношения. Элемент <cmr-field> размещается внутри элемента <ejb-relationship-role>:

```
<ejb-relationship-role>
    <ejb-relationship-role-name>
        Customer-has-an-Address
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <ejb-name>CustomerEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <cmr-field-name>homeAddress</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
```

EJB 2.0 требует, чтобы значение элемента <cmr-field-name> начиналось со строчной буквы. Для каждого поля отношения, определяемого элементом <cmr-field>, класс компонента должен включать пару соответствующих абстрактных методов доступа. Один метод в этой паре должен быть определен с именем set<*cmr-field-name*>(), где первый символ <cmr-field-name> преобразован к верхнему регистру. Другой метод определяется в виде get<*cmr-field-name*>(), также с первым символом <cmr-field-name> в верхнем регистре. В следующем примере элемент <cmr-field-name> содержит значение homeAddress, которое соответствует методам getHomeAddress() и setHomeAddress(), определенным в классе CustomerBean:

```
// код класса компонента
public abstract void setHomeAddress(AddressLocal address);
public abstract AddressLocal getHomeAddress();

// объявление XML-дескриптора развертывания
<cmr-field>
    <cmr-field-name>homeAddress</cmr-field-name>
</cmr-field>
```

Элемент <cascade-delete> требует каскадного удаления. Он может использоваться с отношениями типа «один-к-одному» или «один-ко-многим» и всегда объявляется в виде пустого элемента: <cascade-delete/>. Данный элемент указывает, что время жизни одного объектного компонента в определенном отношении зависит от времени жизни другого компонента, участвующего в этом отношении. Далее показано, как следует изменить объявление отношения для компонентов Customer и Address, чтобы получить каскадное удаление:

```
<relationships>
    <ejb-relation>
```

```
        <ejb-relationship-role>
            <multiplicity>One</multiplicity>
            <role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </role-source>
            <cmr-field>
                <cmr-field-name>homeAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <multiplicity>One</multiplicity>
            <cascade-delete/>
            <role-source>
                <dependent-name>Address</dependent-name>
            </role-source>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

В результате такого объявления компонент Address будет автоматически удаляться при удалении ссылающегося на него компонента Customer.

Описание сборки компонента

К этому моменту мы уже сказали почти все, что можно сказать о самом компоненте. Мы подошли к концу элемента <enterprise-beans> и теперь готовы описать, как компоненты собираются в приложение. То есть мы готовы рассказать о другом важном элементе внутри элемента <ejb-jar> – об элементе <assembly-descriptor>.

Элемент <assembly-descriptor> не относится к обязательным, хотя трудно представить компонент, который может быть успешно развернут без <assembly-descriptor>. Говоря, что <assembly-descriptor> необязательный, в действительности мы имеем в виду, что разработчик, чьей единственной обязанностью является создание компонентов (например, некто, разрабатывающий компоненты для использования третьей стороной и не участвующий в развертывании компонентов), может опустить эту часть дескриптора развертывания. Дескриптор будет правильным и без <assembly-descriptor>, но кто-то почти наверняка должен будет внести информацию о сборке перед тем, как компонент можно будет развернуть.

Элемент <assembly-descriptor> служит трем целям: он описывает атрибуты транзакции для методов компонента, описывает логические роли безопасности, используемые в правах доступа к методам, и определяет права доступа к методам (т. е. каким ролям какой метод разрешается вызвать). В конце <assembly-descriptor> может содержать

три вида элементов, каждый из которых сам по себе довольно сложен. Вот они:

<container-transaction> *(несколько или ни одного)*

Объявляет, какие атрибуты транзакции к каким методам будут применяться. Он содержит необязательный элемент <description>, один или несколько элементов <method> и ровно один элемент <trans-attribute>. Объектные компоненты должны содержать объявления <container-transaction> для всех методов удаленного и внутреннего интерфейсов. Сеансовые компоненты, самостоятельно управляющие своими транзакциями, не будут включать объявление <container-transaction>. Этот элемент рассматривается более подробно в следующем разделе.

<security-role> *(несколько или ни одного)*

Задает роли безопасности, учитываемые при доступе к компоненту. Эти роли используются в элементе <method-permission>. Элемент <security-role> содержит необязательное описание и один элемент <role-name>. Этот элемент вместе с элементом <method-permission> более подробно описан далее в разделе «Определение ролей безопасности и прав доступа к методам».

<method-permission> *(несколько или ни одного)*

Указывает, каким ролям безопасности разрешается вызывать один или несколько методов компонента. Он содержит необязательный элемент <description>, один или несколько элементов <role-name> и один или несколько элементов <method>. Он рассматривается более подробно в разделе «Определение ролей безопасности и прав доступа к методам» вместе с элементом <security-role>.

Элементы <container-transaction> и <method-permission> полагаются на возможность идентифицировать определенные методы. Идентификация может оказаться трудной задачей в связи с такой особенностью языка Java, как перегрузка методов. Для идентификации методов внутри этих тегов предназначен элемент <method>. Он описан подробно далее в разделе «Идентификация отдельных методов».

Задание атрибутов транзакции компонента

Атрибуты транзакции для всех компонентов, определенных в дескрипторе развертывания, задаются в элементах <container-transaction>. Элемент <container-transaction> связывает один или несколько методов компонента с одним атрибутом транзакции, поэтому в каждом <container-transaction> определяется один атрибут транзакции и один или несколько методов компонента.

Элемент <container-transaction> включает в себя единственный элемент <trans-attribute>, который может иметь одно из шести значений: NotSupported, Supports, Required, RequiresNew, Mandatory и Never. Это атри-

буты транзакции, рассмотренные нами в главе 14. Кроме <trans-attribute> элемент <container-transaction> содержит один или несколько элементов <method>.

Сам элемент <method> содержит по крайней мере два подэлемента: элемент <ejb-name>, в котором задается имя компонента, и элемент <method-name>, определяющий подмножество методов компонента. Значением <method-name> может быть имя метода или звездочка (*), которая действует как знак подстановки (wildcard) для всех методов компонента. Гораздо больше сложностей связано с обработкой перегрузки и других специальных случаев, но пока сказанного достаточно, все остальное мы рассмотрим позже.

Далее приводится пример, иллюстрирующий, как обычно используется элемент <container-transaction>. Давайте снова посмотрим на компонент Cabin, который у нас везде выступает в роли примера. Предположим, что нам необходимо указать для метода create() атрибут транзакции Mandatory, а все другие методы будут использовать атрибут Required:

```
<container-transaction>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>create</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
</container-transaction>
```

В первом элементе <container-transaction> у нас есть единственный элемент <method>, в котором группирующий символ (*) применяется для ссылки на все методы компонента Cabin. Для этих методов мы устанавливаем атрибут транзакции Required. У нас также есть второй элемент <container-transaction>, в котором определен единственный метод компонента Cabin – create(). Для этого метода мы устанавливаем атрибут транзакции Mandatory. Эта установка отменяет установку группирующего символа. В элементах <container-transaction> объявления отдельных методов всегда замещают более общие объявления.

Следующим методам необходимо назначать атрибуты транзакции для каждого компонента, объявленного в дескрипторе развертывания.

Для объектных компонентов:

- Все прикладные методы, определенные в удаленном интерфейсе (и всех базовых интерфейсах)

- Конструирующие методы, определенные во внутреннем интерфейсе
- Поисковые методы, определенные во внутреннем интерфейсе
- Внутренние методы, определенные во внутреннем интерфейсе (EJB 2.0)
- Удаляющие методы, определенные в интерфейсах `EJBHome` и `EJBObject`

Для сеансовых компонентов:

- Все прикладные методы, определенные в удаленном интерфейсе (и всех базовых интерфейсах)

В сеансовых компонентах атрибуты транзакции имеют только прикладные методы; конструирующие и удаляющие методы сеансовых компонентов не имеют атрибутов транзакции.

В EJB 2.0 методы `ejbSelect()` не имеют своих собственных атрибутов транзакции. Методы `ejbSelect()` всегда распространяют транзакцию вызывающих их методов.

Определение ролей безопасности и прав доступа к методам

Для того чтобы определить логические роли безопасности и указать, какие роли могут вызывать отдельные методы компонента, предназначены два элемента. Элемент `<security-role>` может содержать необязательный элемент `<description>` и единственный элемент `<role-name>`, задающий имя. Элемент `<assembly-descriptor>` может содержать любое количество элементов `<security-role>`.

Важно понять, что определяемые здесь имена ролей безопасности не наследуются из какой-либо области безопасности. Это логические имена ролей безопасности, они представляют собой просто метки, которые во время развертывания могут быть сопоставлены с реальными ролями безопасности целевого окружения. Так, следующие объявления `<security-role>` определяют две роли – `everyone` и `administrator`:

```
<security-role>
    <description>
        Эта роль представляет всех, кому разрешено чтение/запись для
существующемго компонента Cabin EJBs.
    </description>
    <role-name>everyone</role-name>
</security-role>
<security-role>
    <description>
        Эта роль представляет администратора или менеджера, которым разрешено
        создавать новые экземпляры компонента Cabin. Эта роль может также
        участвовать в роли «everyone».
    </description>
```

```
        <role-name>administrator</role-name>
    </security-role>
```

Эти имена ролей могут не существовать в окружении, в котором будут развернуты компоненты. Не существует никаких предустановок, которые давали бы роли everyone меньше (или больше) привилегий, чем роли administrator. Связывание одной или нескольких ролей целевого окружения с логическими ролями в дескрипторе развертывания возлагается на управляющего развертыванием. Так, например, он может выяснить, что в целевом окружении имеется две роли – DBA (администратор базы данных) и CSR (представитель службы по работе с клиентами), которые соответствуют ролям administrator и everyone, определенным в элементе <security-role>.

Связывание ролей с методами

Роли безопасности сами по себе не представляют большой ценности, пока мы не определим, что этим ролям разрешается делать. И здесь на сцену выходит элемент <method-permission>, связывающий роли безопасности с методами в удаленном и внутреннем интерфейсах компонента. Разрешение методов – достаточно гибкое объявление, позволяющее устанавливать между методами и ролями отношение «многие-ко-многим». <method-permission> содержит необязательный элемент <description>, один или несколько элементов <method> и один или несколько элементов <role-name>. Имена, указанные в элементах <role-name>, соответствуют ролям, определенным в элементах <security-role>.

Приведем один из способов установки разрешения методов для компонента Cabin:

```
<method-permission>
    <role-name>administrator</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
<method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>getDeckLevel</method-name>
    </method>
</method-permission>
```

В этом примере роль administrator имеет доступ ко всем методам компонента Cabin. Роль everyone имеет доступ только к методу getDeckLevel(), она не может обратиться ни к какому другому методу компонента Cabin. Обратите внимание, что указанные разрешения методов группируются в виде объединения. Например метод getDeckLevel() до-

ступен и роли administrator, и everyone, что представляет собой объединение разрешений, объявленных в описателе. Еще раз обратим ваше внимание на то, что мы все еще не знаем, что обозначают administrator и everyone. Они определяются человеком, разворачивающим компонент, который должен связать эти логические роли безопасности с реальными ролями безопасности, заданными в целевом окружении.

Все методы, определенные в удаленном или внутреннем интерфейсе и во всех базовых интерфейсах, включая методы, определенные в интерфейсах EJBObject и EJBHome, могут быть связаны с ролями безопасности в элементах <method-permission>. Любой метод, который не был включен в них, не будет доступен никакой роли безопасности.

EJB 2.0: Непроверяемые методы

В EJB 2.0 набор методов может быть обозначен как *непроверяемые* (*unchecked*). Это означает, что перед их вызовом не выполняется проверка прав доступа. Непроверяемый метод может быть вызван любым клиентом независимо от того, какой ролью он пользуется.

Указать, что метод или методы являются непроверяемыми, можно с помощью элемента <method-permission> и замены элемента <role-name> на пустой элемент <unchecked>:

```
<method-permission>
    <unchecked/>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
    <method>
        <ejb-name>CustomerEJB</ejb-name>
        <method-name>findByPrimaryKey</method-name>
    </method>
</method-permission>
<method-permission>
    <role-name>administrator</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
```

Это объявление сообщает нам, что все методы компонента Cabin, а также метод findByPrimaryKey() компонента Customer относятся к непроверяемым. Хотя второй элемент <method-permission> дает администратору разрешение обращаться ко всем методам компонента Cabin, это объявление замещается разрешением непроверяемого метода. Разрешения непроверяемых методов всегда замещают все остальные разрешения методов.

EJB 2.0: Идентификатор безопасности runAs

Кроме определения принципалов (principals), имеющих доступ к методам компонента, управляющий развертыванием может определить принципал *runAs* для всего компонента. Идентификатор безопасности *runAs* первоначально был определен в EJB 1.0, но затем был отменен в EJB 1.1. Он был повторно введен в EJB 2.0 и изменился так, чтобы его было проще реализовывать производителям.

В то время как элементы `<method-permission>` указывают, какие принципалы имеют доступ к методам компонента, элемент `<security-identity>` определяет, под каким принципалом будет исполняться данный метод. Другими словами, принципал *runAs* используется в качестве идентификатора компонента, когда тот пытается вызывать методы других компонентов. Этот идентификатор не обязательно является тем же идентификатором, который в настоящее время обращается к компоненту.

Например, следующие элементы дескриптора развертывания объявляют, что к методу `create()` может обращаться только `JimSmith`, но компонент Cabin всегда будет выполняться под идентификатором безопасности `Administrator`:

```
<enterprise-beans>
...
    <entity>
        <ejb-name>EmployeeService</ejb-name>
        ...
        <security-identity>
            <run-as>
                <role-name>Administrator</role-name>
            </run-as>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
<assembly-descriptor>
    <security-role>
        <role-name>Administrator</role-name>
    </security-role>
    <security-role>
        <role-name>JimSmith</role-name>
    </security-role>
    ...
    <method-permission>
        <role-name>JimSmith</role-name>
        <method>
            <ejb-name>CabinEJB</ejb-name>
            <method-name>create</method-name>
        </method>
    </method-permission>
```

```
    ...
</assembly-descriptor>
```

Компонент выполнится под идентификатором вызывающего объекта, – роль <security-identity> содержит указывающий на это единственный пустой элемент, <use-caller-identity/>. Следующие объявления задают, что компонент Cabin всегда будет выполняться под идентификатором вызывающего объекта. Так, если Джим Смит вызывает метод create(), то компонент будет выполняться под идентификатором безопасности JimSmith:

```
<enterprise-beans>
...
    <entity>
        <ejb-name>EmployeeService</ejb-name>
        ...
        <security-identity>
            <use-caller-identity/>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
```

Элемент <security-identity> одинаково применяется и к объектным компонентам, и к сеансовым компонентам без состояния. Однако у компонентов, управляемых сообщениями, есть идентификатор *runAs*. Они никогда не выполняются под идентификатором вызывающего объекта просто потому, что у них нет «вызывающего объекта». Асинхронные сообщения JMS, которые обрабатываются компонентами, управляемыми сообщениями, не считаются вызовами, а посылающие их клиенты JMS не связаны с этими сообщениями. При отсутствии идентификатора вызывающего объекта, подлежащего распространению, компоненты, управляемые сообщениями, должны всегда иметь заданный идентификатор безопасности *runAs*.

EJB 2.0: Исключающий список

Последним элементом <assembly-descriptor> является необязательный элемент <exclude-list>. Элемент <exclude-list> включает в себя <description> и ряд элементов <method>. Каждый метод, перечисленный в <exclude-list>, должен рассматриваться как невызываемый (uncallable). Это означает, что управляющий развертыванием должен установить разрешения безопасности для этих методов так, чтобы все вызовы, приходящие от любого клиента, отвергались. Удаленные клиенты должны получать исключение java.rmi.remoteException, а локальные клиенты – исключение javax.ejb.AccessLocalException:

```
<ejb-jar>
    <enterprise-beans>
```

```
        <entity>
            <ejb-name>CabinEJB</ejb-name>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <exclude-list>
            <method>
                <ejb-name>CabinEJB</ejb-name>
                <method-name>getDeckLevel</method-name>
            </method>
            <method>
                ...
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

Описание отдельных методов

Элемент <method> помещается в элементы <method-permission> и <container-transaction> для определения отдельной группы методов определенного компонента. Элемент <method> всегда содержит элемент <ejb-name>, указывающий имя компонента, и элемент <method-name>, определяющий метод. Он также может включать элемент <description>, элементы <method-params>, определяющие, какие параметры будут использоваться в методе для того, чтобы разрешить неоднозначность с перегруженными методами, и элемент <method-intf>, определяющий, принадлежит ли метод внутреннему, удаленному, локальному внутреннему или локальному интерфейсу компонента. Этот последний элемент учитывает такую возможность, при которой в нескольких интерфейсах могут использоваться одинаковые имена методов.

Объявления со знаком подстановки

Имя метода в элементе <method> может просто содержать знак подстановки (*). Знак подстановки применим ко всем методам во внутреннем и удаленном интерфейсах компонента. Например:

```
<method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>*</method-name>
</method>
```

Хотя иногда возникает желание объединить знак подстановки с другими символами, делать это не следует. Например, запись get* содержит ошибку. Символ звездочки (*) может использоваться только самостоятельно.

Именованные объявления методов

Именованные объявления применяются ко всем методам, определенным в удаленном и внутреннем интерфейсах компонента, имеющих указанные имена. Например:

```
<method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>create</method-name>
</method>
<method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>getDeckLevel</method-name>
</method>
```

Эти объявления применяются ко всем методам с данным именем в обоих интерфейсах. Они не различают перегруженные методы. Например, если внутренний интерфейс для компонента Cabin изменяется так, что в нем появляются три перегруженных метода `create()`, как показано здесь, предыдущее объявление `<method>` будет относиться ко всем трем методам:

```
public interface CabinHome javax.ejb.EJBHome {
    public Cabin create() throws CreateException, RemoteException;
    public Cabin create(int id) throws CreateException, RemoteException;
    public Cabin create(int id, Ship ship, double [][] matrix)
        throws CreateException, RemoteException;
    ...
}
```

Объявления отдельных методов

Объявления отдельных методов используют элемент `<method-params>` для точного указания на определенный метод, перечисляя его параметры, что позволяет проводить различия между перегруженными методами. Элемент `<method-params>` содержит несколько необязательных элементов `<method-param>`, которые соответствуют, в порядке следования, каждому типу параметра (включая многомерные массивы), объявленному в методе. Для указания метода, не содержащего параметры, используйте элемент `<method-params>` без вложенных в него элементов `<method-param>`.

Например, давайте снова взглянем на наш компонент Cabin, во внутренний интерфейс которого мы добавили несколько перегруженных методов `create()`. Здесь имеются три элемента `<method>`, каждый из которых однозначно определяет один из методов `create()`, перечисляя его параметры:

```
<method>
    <description>Method: public Cabin create(); </description>
```

```
    <ejb-name>CabinEJB</ejb-name>
    <method-name>create</method-name>
    <method-params></method-params>
</method>
<method>
    <description>Method: public Cabin create(int id);</description>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>create</method-name>
    <method-params>
           <method-param>int</method-param>
    </method-params>
</method>
<method>
    <description>
         Method: public Cabin create(int id, Ship ship, double [][] matrix);
    </description>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>create</method-name>
    <method-params>
           <method-param>int</method-param>
           <method-param>com.titan.ship.Ship</method-param>
           <method-param>double [][]</method-param>
    </method-params>
</method>
```

Разделение удаленных, внутренних и локальных интерфейсов

Нам осталось рассмотреть еще одну проблему. Во внутреннем интерфейсе, локальном внутреннем интерфейсе, удаленном интерфейсе и локальном интерфейсе могут применяться методы с одинаковыми именами. Для разрешения этой неоднозначности можно в качестве модификатора добавить к объявлению метода элемент `<method-intf>`. Для элемента `<method-intf>` допустимы четыре значения: `Remote`, `Home`, `LocalHome` и `Local`. В действительности маловероятно, чтобы опытный разработчик давал одинаковые имена методов и во внутреннем, и в удаленном интерфейсе. Это привело бы к сильно запутанному коду. Однако необходимо быть готовыми встретить одинаковые имена в локальном и удаленном интерфейсах или внутреннем и локальном внутреннем интерфейсах. Также вероятно, что элемент `<method-intf>` вам потребуется в объявлении, содержащем символ подстановки. Например, в следующем объявлении указаны все методы в удаленном интерфейсе компонента Cabin:

```
<method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>*</method-name>
    <method-intf>Remote</method-intf>
</method>
```

Все эти стили объявлений методов могут применяться в любой комбинации внутри любого элемента, содержащего элемент <method>. Элементы <method-permission> объединены для формирования объединения из разрешений типа «роль-метод». Например, в следующем листинге первый элемент <method-permission> объявляет, что administrator имеет доступ к внутренним методам компонента Cabin (конструирующим и поисковым методам). Второй <method-permission> указывает, что everyone имеет доступ к методу findByPrimaryKey(). Это означает, что обе роли (everyone и administrator) имеют доступ к методу findByPrimaryKey():

```
<method-permission>
    <role-name>administrator</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
        <method-intf>Home</method_intf>
    </method>
</method-permission>
<method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>findByPrimaryKey</method-name>
    </method>
</method-permission>
```

Файл ejb-jar

Формат файла JAR является независимым от платформы форматом для совместного сжатия, упаковки и распространения нескольких файлов. Основанные на формате файла ZIP и стандартах сжатия ZLIB, пакеты и средства JAR (Java archive – архив Java) были первоначально разработаны для более эффективной загрузки апплетов Java. Однако как механизм упаковки формат файла JAR оказался удобным способом для совместной упаковки компонентов и другого программного обеспечения для распространения третьими лицами. Оригинальная компонентная архитектура JavaBeans при упаковке полагается на файлы JAR, то же самое справедливо и для Enterprise JavaBeans. Цель применения формата файла JAR состоит в том, чтобы упаковать в один файл все классы и интерфейсы, связанные с одним или несколькими компонентами, включая дескриптор развертывания.

Файл JAR создается при помощи специализированного средства или с помощью утилиты *jar*, которая является частью пакета разработки Java 2, Standard Edition. Файл *ejb-jar* содержит:

- XML-дескриптор развертывания
- Классы компонента

- Удаленный и внутренний интерфейсы
- Класс первичного ключа
- Вспомогательные классы и интерфейсы

XML-дескриптор развертывания должен быть расположен в каталоге *META-INF/ejb-jar.xml* и должен содержать полную информацию по развертыванию всех компонентов, находящихся в файле *ejb-jar*. Для каждого компонента, объявленного в XML-дескрипторе развертывания, файл *ejb-jar* должен содержать его компонентный класс, удаленный и внутренний интерфейсы и вспомогательные классы и интерфейсы. Вспомогательные классы и интерфейсы достаточно распространены и обычно включают в себя специфичные для данного приложения исключения, прикладные интерфейсы, другие базовые типы и вспомогательные объекты, используемые компонентом. Например, в файл *ejb-jar* для компонента TravelAgent мы могли бы поместить прикладное исключение `IncompleteConversationalState` и классы `Ticket` и `CreditCard`, а также удаленные и внутренний интерфейсы других компонентов (таких как Customer и ProcessPayment), на которые ссылается компонент TravelAgent.[1]

Упаковать компонент в файл JAR можно посредством утилиты командной строки *jar*. Далее приведен пример ее использования для упаковки компонента Cabin из главы 4:

```
\dev % jar cf cabin.jar com/titan/cabin/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\cabin\*.class META-INF\ejb-jar.xml
```

Возможно, сначала придется создать каталог *META-INF* и скопировать в него *ejb-jar.xml*. Ключ *c* указывает утилите *jar* создать новый файл JAR, который будет содержать файлы, указанные в последующих параметрах. Он также сообщает утилите *jar* о необходимости направить итоговый файл JAR на стандартный вывод. Ключ *f* перенаправляет стандартный вывод в новый файл, заданный вторым параметром (*cabin.jar*). Необходимо точно воспроизвести порядок символов ключей и параметров командной строки. Об утилите *jar* и пакете `java.util.zip` подробно рассказано в книгах «Java in a Nutshell» (Java. Справочник) Дэвида Флэнагана (David Flanagan) и «Learning Java» (Изучаем Java) Пата Нимайера (Pat Niemeier) и Джонатана Кнудсена (Jonathan Knudsen), выпущенных издательством O'Reilly.

[1] Спецификация EJB 1.1 также позволяет задавать имена удаленных и внутренних интерфейсов компонентов в атрибуте `Class-Path` манифеста вместо включения их в файл JAR. Использование элемента `Class-Path` в манифесте файла JAR подробно рассматривается в спецификации Java 2, Standard Edition.

Утилита *jar* создала в каталоге *dev* файл *cabin.jar*. Ознакомиться с содержимым файла JAR можно при помощи любого стандартного приложения ZIP (WinZip, PKZIP и т. д.) или посредством команды *jar tvf cabin.jar*.

Файл client-jar

В EJB 1.1 также разрешен файл *client-jar*, включающий только интерфейсы и классы, необходимые клиентскому приложению для доступа к компоненту. Он включает удаленный и внутренний интерфейсы, первичный ключ и любые вспомогательные типы, экспортируемые клиентом, такие как прикладные исключения. Спецификация не указывает, как этот файл должен передаваться клиенту, что конкретно он должен содержать и как он упаковывается совместно с файлом *ejb-jar*. Другими словами, файл *client-jar* в EJB является понятием достаточно сильно зависимым от конкретного производителя.

17

Java 2, Enterprise Edition

Спецификация Java 2, Enterprise Edition (J2EE) определяет платформу для разработки интернет-приложений, которая включает компоненты Enterprise JavaBeans, сервлеты и серверные страницы Java (JavaServer Pages, JSP). Продукты J2EE – это серверы приложений, предоставляющие законченную реализацию технологий EJB, сервлетов и JSP. Кроме того, J2EE описывает, как эти технологии должны взаимодействовать, чтобы предоставить законченное решение. Для лучшего понимания J2EE необходимо познакомиться с сервлетами и JSP и проследить связь между этими технологиями и Enterprise JavaBeans.

Образно говоря, J2EE предоставляет два вида «клея», облегчающие взаимодействие компонентов. Мы уже видели оба эти типа клея. Во-первых, для того чтобы стандартизировать способ, которым компоненты ищут необходимые им ресурсы, используется контекст имен JNDI (ENC), с которым мы знакомы по обсуждению компонентов. В данной главе мы кратко рассмотрим, как сервлеты, JSPs и даже некоторые клиенты могут использовать ENC для нахождения ресурсов. Во-вторых, применение дескрипторов развертывания (в частности, использование XML для определения языка для дескрипторов развертывания) было расширено на сервлеты и JSP. Сервлеты Java и страницы JSP могут быть упакованы с помощью дескрипторов развертывания, которые определяют их отношения со своим окружением. Дескрипторы развертывания также используются для описания сборки нескольких компонентов (assemblies) в прикладные программы.

Сервлеты

Спецификация сервлетов определяет серверную компонентную модель, которая может быть реализована производителями веб-серверов. Сервлеты предоставляют простой, но мощный API для генерации динамических веб-страниц. (Хотя сервлеты могут применяться для самых разных протоколов типа «запрос-ответ», они преимущественно используются для обработки HTTP-запросов веб-страниц.)

Сервлеты разрабатываются таким же образом, как и компоненты. Они представляют собой классы Java, которые расширяют класс базового компонента и имеют дескриптор развертывания. После того как сервлет создан и упакован в файл JAR, он может быть развернут на веб-сервере. Во время развертывания сервлет назначается на обработку запросов указанной веб-страницы или помогает другим сервлетам обрабатывать запросы страниц. Следующий сервлет, например, может быть назначен на обработку всех запросов к странице *helloworld.html* на веб-сервере:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse
response)
        throws ServletException,java.io.IOException {

    try {
        ServletOutputStream writer = response.getWriter();
        writer.println("<HTML><BODY>");
        writer.println("<h1>Hello World!!</h1>");
        writer.println("</BODY></HTML>");
    } catch(Exception e) {
        // Обработка исключений
    }
    ...
}
```

Когда броузер посылает запрос страницы на веб-сервер, сервер перенаправляет запрос соответствующему экземпляру сервлета, вызывая его метод `doGet()`.[1] Сервлету передается информация о запросе через объект `HttpServletRequest`, а для ответа на запрос он может использовать объект `HttpServletResponse`. Этот простой сервлет посылает обратно броузеру короткий HTML-документ, содержащий текст «Hello World». На рис. 17.1 показано, как запрос посылается броузером и обрабатывается сервлетом, выполняющимся на веб-сервере.

[1] Интерфейс `HttpServlet` также содержит метод `doPost()`, обрабатывающий запросы от форм.

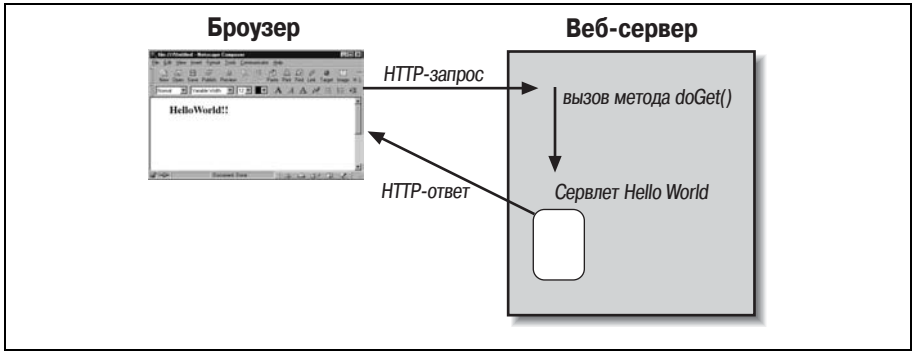

Рис. 17.1. Сервлет, обслуживающий HTTP-запрос

Сервлеты похожи на сеансовые компоненты тем, что все они занимаются обслуживанием и могут через JDBC непосредственно обращаться к ресурсам с данными (например, к базам данных), но они не представляют постоянных данных. Однако сервлеты не поддерживают транзакции, управляемые контейнером, и не состоят из прикладных методов. Сервлеты имеют дело с узкоспециальными (обычно HTTP) запросами и отвечают на них записью в выходной поток.

Спецификация сервлетов довольна обширная и мощная, но, вместе с тем, простая и изящная. Она представляет собой мощную серверную компонентную модель. Подробнее о сервлетах можно узнать из книги Джейсона Хантера (Jason Hunter) и Уильяма Кроуфорда (William Crawford) «Java Servlet Programming», O'Reilly, 2001.[1]

JavaServer Pages

Серверные страницы Java представляют собой расширение компонентной модели сервлетов, которая упрощает процесс динамического создания HTML. По существу, JSP позволяет включать непосредственно в страницы HTML код Java в качестве языка сценариев. В J2EE код Java из страницы JSP может обращаться к JNDI ENC точно так же, как и из сервлета. Фактически JSP-страницы (текстовые документы) транслируются и компилируются в сервлеты Java, которые затем выполняются на веб-сервере точно так же, как и любой другой сервлет – некоторые серверы выполняют трансляцию автоматически во время выполнения. JSP может применяться и для динамической генерации XML-документов.

О JSP можно больше узнать из книги «JavaServer Pages™» (Серверные страницы Java) Ханса Бергстена (Hans Bergsten), O'Reilly.

[1] Джейсон Хантер, Уильям Кроуфорд «Программирование Java-сервлетов», 2-е издание, издательство «Символ-Плюс», 2002.

Веб-компоненты и EJB

Сервлеты совместно с JSP предоставляют мощную платформу для динамической генерации страниц. Сервлеты и JSP, имеющие общее название *веб-компоненты* (*web components*), могут обращаться к ресурсам, таким как JDBC, и компонентам EJB. Благодаря тому что веб-компоненты имеют доступ к базам данных через JDBC, они могут предоставить мощную платформу для электронной коммерции, позволяя компании выставить ее бизнес-системы в Интернет через интерфейс HTML. HTML имеет несколько преимуществ перед традиционными клиентскими приложениями, написанными на Java или каком-либо другом языке. Наиболее важные преимущества связаны с распространением и сетевыми устройствами защиты. Традиционные клиенты требуют распространения и установки на клиентских машинах, следствием чего является одно серьезное ограничение: они требуют выполнения дополнительной работы по установке и обслуживанию. Для уменьшения количества хлопот при инсталляции могут применяться динамически загружаемые апплеты, но они имеют свои ограничения, такие как «песочница» и тяжеловесная загрузка. В противоположность им HTML чрезвычайно легок, не требует предварительной инсталляции и ему не свойственны ограничения безопасности. Кроме того, интерфейсы HTML могут быть изменены и расширены на их источнике без необходимости модификации клиентов.

Брандмауэры (сетевые устройства защиты) представляют еще одну серьезную проблему для электронной коммерции. Протокол HTTP, с помощью которого запрашиваются и передаются веб-страницы, может без затруднений проходить через большинство брандмауэров, но другие протоколы, такие как IIOP и JRMP, не могут. Это ограничение кажется серьезным препятствием на пути к успеху систем распределенных объектов, которые должны поддерживать доступ со стороны анонимных клиентов, поскольку это означает, что приложения на основе распределенных объектов вообще не могут быть созданы на клиентской платформе, которая может иметь произвольную конфигурацию брандмауэра. У протокола HTTP нет такого ограничения, т. к. практически все брандмауэры разрешают беспрепятственную передачу по протоколу HTTP.

Проблемы с распространением и брандмауэрами привели индустрию EJB к принятию, по большей части, архитектуры, основанной на совместном использовании веб-компонентов (сервлетов/JSP) и компонентов Enterprise JavaBeans. В то время как веб-компоненты содержат логику представления для создания веб-страниц, EJB предоставляет мощный транзакционный промежуточный уровень для прикладной логики. Веб-компоненты обращаются к компонентам EJB, используя тот же самый API, что и прикладные клиенты. Каждая технология занимается тем, что она умеет делать лучше всего: сервлеты и JSP являются незаменимыми компонентами для динамического создания

HTML, тогда как EJB представляет собой превосходную платформу для транзакционной прикладной логики. На рис. 17.2 показано, как работает эта архитектура.

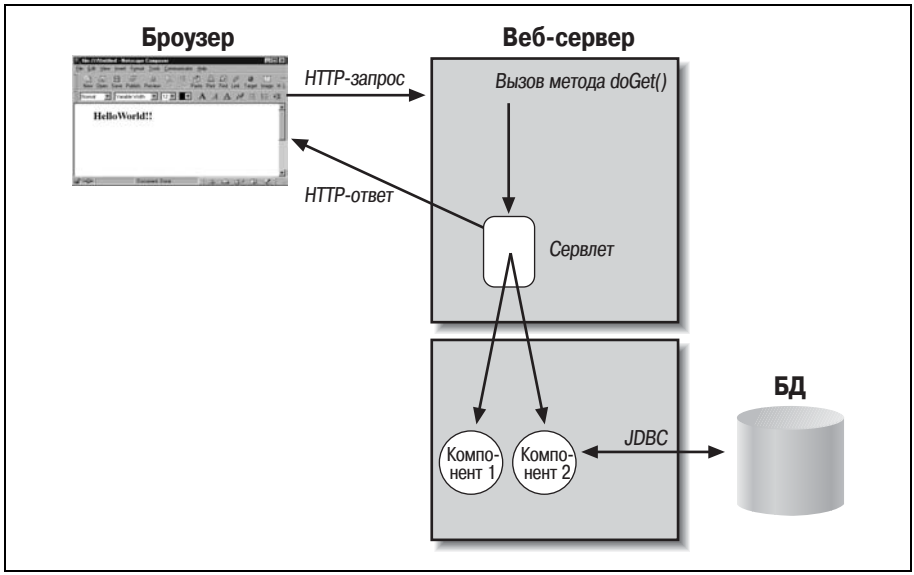

Рис. 17.2. Совместное использование сервлетов/JSP и EJB

Эта архитектура «веб-компонент–EJB» так широко распространена, что вопрос «Нужна ли объединенная платформа?», на который спецификация J2EE должна была найти ответ, отпадает сам собой. Спецификация J2EE определяет единую прикладную серверную платформу, обеспечивающую взаимодействие между сервлетами, JSP и EJB. J2EE важна, поскольку она предоставляет спецификацию, описывающую взаимодействие веб-компонентов с компонентами EJB, делая продукты от разных производителей, поддерживающих обе компонентные модели, более переносимыми.

J2EE заполняет пробелы

Спецификация J2EE пытается заполнить пробелы, существующие между веб-компонентами и Enterprise JavaBeans, определяя, как следует объединить эти технологии, чтобы сформировать законченную платформу.

Один из способов, которыми J2EE может быть полезен, состоит в создании согласованной программной модели, включающей веб-компоненты и компоненты EJB, с применением JNDI ENC и XML-дескрипторов развертывания. Сервлет в J2EE может обращаться к объектам JDBC `DataSource`, элементам окружения и ссылкам на компоненты EJB

через JNDI ENC точно таким же образом, каким используют JNDI ENC компоненты EJB. Для поддержки JNDI ENC у веб-компонентов есть свои собственные XML-дескрипторы развертывания, в которых объявлены элементы для JNDI ENC (<ejb-ref>, <resource-ref>, <env-entry>), а также роли безопасности и другие элементы, специфичные для веб-компонентов. В J2EE веб-компоненты упаковываются вместе со своими XML-дескрипторами развертывания и разворачиваются в файлах JAR, имеющих расширение *.war* (от *w*eb *ar*chive). Использование в веб-компонентах JNDI ENC, дескрипторов развертывания и файлов JAR делает их совместимыми с программной моделью EJB и унифицирует всю платформу J2EE.

JNDI ENC значительно упрощает обращение веб-компонентов к Enterprise JavaBeans. Разработчик веб-компонента не должен беспокоиться о расположении компонентов в сети. Сервер сам будет сопоставлять элементы <ejb-ref>, перечисленные в дескрипторе развертывания, с компонентами во время развертывания.

Дополнительно производители J2EE могут разрешить веб-компонентам доступ к локальным интерфейсам компонентов EJB 2.0. Это имеет смысл, если веб-компонент и компонент EJB расположены в одной виртуальной машине Java, т. к. использование семантики Java RMI-IIOP может улучшить производительность. Ожидается, что большинство производителей J2EE будут поддерживать эту возможность.

Кроме того, JNDI ENC поддерживает доступ к объекту javax.jta.UserTransaction, как это имеет место в EJB. Объект UserTransaction позволяет веб-компонентам управлять транзакциями явно. Контекст транзакции должен быть распространен на все компоненты, к которым выполняется обращение внутри зоны транзакции (согласно атрибуту транзакции метода компонента). Файл *.war* может содержать несколько сервлетов и JSP-документов, совместно использующих XML-дескриптор развертывания.

J2EE определяет также файл *.ear* (*e*nterprise *ar*chive), который представляет собой файл JAR, служащий для упаковки вместе файлов EJB JAR и файлов JAR с веб-компонентами (файлов *.war*) в одно законченное приложение, называемое приложением J2EE. У приложения J2EE есть свой собственный XML-дескриптор развертывания, указывающий на файлы JAR для компонентов EJB и веб-компонентов (называемые модулями), а также на другие элементы, такие как изображения, описания и т. п. Во время создания приложения J2EE могут быть разрешены взаимозависимости, такие как элементы <ejb-ref> и <ejb-local-ref>, и отредактированы роли безопасности для придания стандартного представления всему веб-приложению. На рис. 17.3 показана структура архивного файла J2EE.

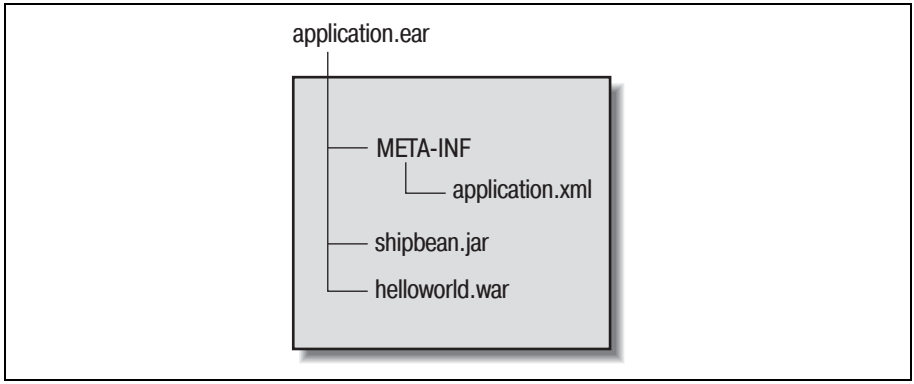

Рис. 17.3. Содержание файла J2EE EAR

Прикладные клиентские компоненты J2EE

В дополнение к интеграции веб-компонентов и компонентов EJB спецификация J2EE представляет совершенно новую компонентную модель: прикладной клиентский компонент. Прикладной клиентский компонент – это Java-приложение, находящееся на машине клиента и обращающееся к компонентам EJB на сервере J2EE. Клиентские компоненты также имеют доступ к JNDI ENC, который действует таким же образом, как и JNDI ENC для веб-компонентов и компонентов EJB. Компонент клиента содержит XML-дескриптор развертывания, в котором объявлены элементы JNDI ENC `<env-entry>`, `<ejb-ref>` и `<resource-ref>` в дополнение к `<description>`, `<display-name>` и `<icon>`, которые могут использоваться для представления клиентского компонента в средстве развертывания.

Клиентский компонент – это просто Java-программа, которая использует JNDI ENC для доступа к свойствам окружения, компонентам EJB и ресурсам (JDBC, JavaMail и т. п.), сделанным доступными сервером J2EE. Клиентские компоненты расположены на машине клиента, а не на сервере J2EE. Ниже приведен чрезвычайно простой компонент:

```
public class MyJ2eeClient {

    public static void main(String [] args) {

        InitialContext jndiCntx = new InitialContext();

        Object ref = jndiCntx.lookup("java:comp/env/ejb/ShipBean");
        ShipHome home = (ShipHome)
            PortableRemoteObject.narrow(ref,ShipHome.class);

        Ship ship = home.findByPrimaryKey(new ShipPK(1));
        String name = ship.getName();
```

```
            System.out.println(name);

        }
    }
```

`MyJ2eeClient` иллюстрирует, как следует писать клиентские компоненты. Обратите внимание, что клиентскому компоненту не требуется использовать зависящий от конкретной сети объект `InitialContext` JNDI. Другими словами, для того чтобы соединиться с сервером J2EE, нам не нужно указывать провайдер услуг. Это действительно сильная сторона прикладного клиентского компонента J2EE: прозрачность расположения. Клиентскому компоненту не обязательно знать точное расположение компонента Ship или выбирать определенный провайдер услуг JNDI. JNDI ENC сам заботится о поиске компонента.

Во время разработки прикладных компонентов создается XML-дескриптор развертывания, в котором определяются элементы JNDI ENC. Во время развертывания средствами J2EE генерируются файлы класса, необходимые для развертывания компонента на клиентских машинах.

Клиентский компонент упаковывается в файл JAR вместе с его XML-дескриптором развертывания и может быть включен в состав приложения J2EE. После того как клиентский компонент включен в дескриптор развертывания приложения J2EE, он может быть упакован в файл EAR вместе с другими компонентами, как показано на рис. 17.4.

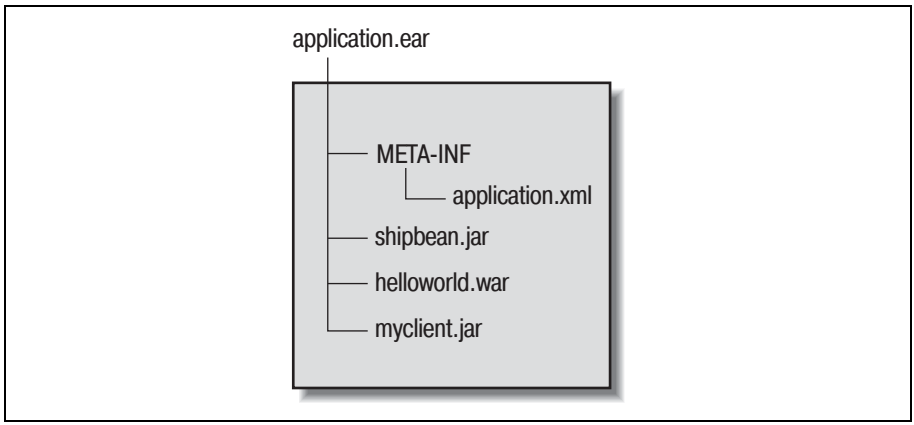

Рис. 17.4. Содержание файла J2EE EAR с прикладным компонентом

Обязательные службы

Спецификация J2EE 1.3 требует, чтобы серверы приложений поддерживали определенный набор протоколов и корпоративных расширений Java. Это требование гарантирует согласованную платформу для

развертывания приложений J2EE. Серверы приложений J2EE должны предоставлять следующие «стандартные» услуги:

Enterprise JavaBeans 2.0

Продукты J2EE должны поддерживать полную спецификацию.

Servlets 2.3

Продукты J2EE должны поддерживать полную спецификацию.

JavaServer Pages 1.2

Продукты J2EE должны поддерживать полную спецификацию.

HTTP и HTTPS

Веб-компоненты в сервере J2EE обслуживают запросы и HTTP, и HTTPS. Продукт J2EE должен уметь обрабатывать запросы HTTP 1.0 и HTTPS (HTTP 1.0 через SSL 3.0) на портах 80 и 443 соответственно.

Java RMI-IIOP

Поддержка Java RMI-IIOP является обязательной. Однако производителем могут использоваться также и другие протоколы в той степени, в которой они совместимы с семантикой Java RMI-IIOP.

Java RMI-JRMP

Компоненты J2EE могут быть клиентами Java RMI-JRMP.

JavaIDL

Веб-компоненты и компоненты EJB должны уметь обращаться к службам CORBA, размещенным за пределами окружения J2EE, с применением JavaIDL, стандартной части платформы Java 2.

JDBC 2.0

J2EE требует поддержки ядра JDBC (JDK 1.3) и некоторых элементов JDBC 2.0 Extention, таких как система имен, пул подключений и поддержка распределенных транзакций.

Интерфейс идентификации и маршрутизации Java (Java Naming and Directory Interface, JNDI) 1.2

Веб-компоненты и компоненты EJB должны иметь доступ к JNDI ENC, который делает доступными объекты `EJBHome`, объекты JTA `UserTransaction`, объекты JDBC `DataSource` и объекты фабрики подключений службы сообщений Java (Java Message Service).

Java. JavaMail 1.2 и JAF 1.0

Продукты J2EE должны поддерживать отправку основных типов сообщений почтовой службы Интернета (протокол не указан) с помощью JavaMail API из компонентов EJB и веб-компонентов. Реализация JavaMail должна поддерживать типы сообщения MIME. JAF – это система активации Java (Java Activation Framework), не-

обходимая для поддержки разных типов MIME и функциональности JavaMail.

Служба сообщений Java (Java Message Service, JMS) 1.0.2

Продукты J2EE должны предоставлять поддержку и для модели передачи сообщений «точка-точка» (p2p), и для модели «издание-подписка» (pub/sub).

Java API для синтаксического разбора XML (Java API for XML Parsing, JAXP) 1.1

Продукты J2EE должны поддерживать JAXP и предоставлять по крайней мере один синтаксический анализатор SAX 2, по крайней мере один синтаксический анализатор DOM 2 и по крайней мере один преобразователь XSLT.

Архитектура соединений J2EE (J2EE Connector Architecture, JCA) 1.0

J2EE должен поддерживать JCA API для всех компонентов и предоставлять полную поддержку для адаптеров ресурсов и свойств транзакций, как определено в JCA.

Служба аутентификации и авторизации Java (Java Authentication and Authorization Service, JAAS) 1.0

Продукты J2EE должны поддерживать использование JAAS, как описано в спецификации JCA. Кроме этого, контейнеры клиентских приложений должны поддерживать средства аутентификации, определенные в спецификации JAAS.

API транзакций Java 1.0.1

Веб-компоненты и компоненты EJB должны иметь доступ к объектам JTA `UserTransaction` через JNDI ENC внутри контекста `"java:comp/UserTransaction"`. Интерфейс `UserTransaction` применяется для явного управления транзакциями.

Собираем все вместе

Для иллюстрации применения платформы J2EE представим себе работу сервера J2EE в системе заказа билетов «Титан». Для создания этой системы мы можем использовать TravelAgent, Cabin, ProcessPayment, Customer и другие компоненты, созданные нами в этой книге, совместно с веб-компонентами, которые могли бы предоставлять HTML-интерфейс.

Веб-компоненты могли бы обращаться к компонентам EJB таким же образом, как любой Java-клиент, используя удаленные и внутренние интерфейсы компонента. Веб-компоненты генерируют HTML для представления системы резервирования.

На рис. 17.5 показана веб-страница, сгенерированная сервлетом или страницей JSP для системы заказа билетов «Титан». Эта веб-страница была сгенерирована веб-компонентами, находящимися на сервере J2EE. Здесь человек, использующий систему заказов, будет проведен через страницу входа в систему, страницу выбора клиента и страницу выбора круиза и будет готовиться к резервированию свободной каюты.

Рис. 17.5. Интерфейс HTML для системы резервирования «Титан»

Список свободных кают был получен из компонента TravelAgent, метод `listAvailableCabins()` которого был вызван сервлетом, сгенерировавшим веб-страницу. На основе списка кают создавался HTML-список в веб-странице, которая была загружена в броузер пользователя. Когда пользователь выбирает каюту и подтверждает выбор, серверу J2EE направляется HTTP-запрос. Сервер J2EE принимает запрос и перенаправляет его сервлету `ReservationServlet`, который вызывает метод `TravelAgent.bookPassage()` для выполнения действительного резервирования. Затем на основе информации о билете, возвращаемой методом `bookPassage()`, создается другая веб-страница, которая отсылается

обратно броузеру пользователя. На рис. 17.6 показано, как взаимодействуют разные компоненты при обработке этого запроса.

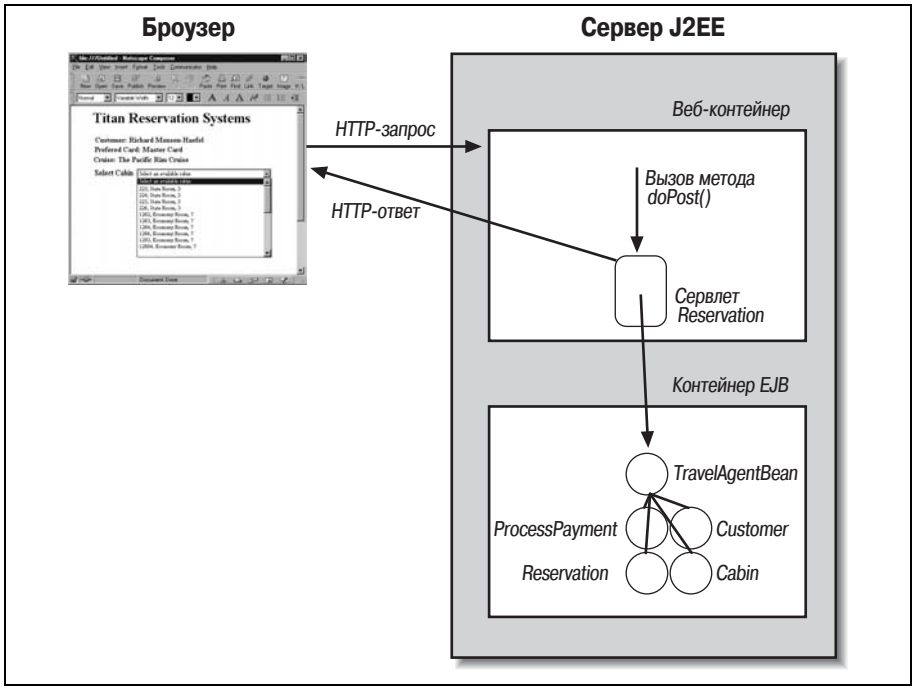

Рис. 17.6. J2EE-система резервирования «Титан»

Будущие расширения

В спецификации J2EE уже намечено несколько направлений усовершенствования следующей версии. Как ожидается, поддержка «веб-служб» в будущем займет большую часть спецификации J2EE, в их состав будут включена поддержка Java API для XML-сообщений (Java API for XML Messaging, JAXM), Java API для XML-реестра (Java API for XML Registry, JAXR) и Java API для XML RPC (JAX-RPC). Поддержка API для привязки XML-данных (XML Data Binding API), который считается более легким в использовании, чем JAXP, также может быть обязательной в будущей версии спецификации.

Кроме того, J2EE может быть расширен, и не исключено, что он потребует поддержки наборов строк JDBC, SQLJ, API для управления и развертывания и, возможно, J2EE SPI, построенный на основе усовершенствований, сделанных с помощью спецификации JCA.

A

Программный интерфейс Enterprise JavaBeans

Это приложение представляет собой краткое руководство по программному интерфейсу (API) Enterprise JavaBeans. Внутри каждого пакета классы организованы в алфавитном порядке.

Пакет javax.ejb

В данном пакете содержится ядро API EJB. Он состоит главным образом из интерфейсов, многие из которых реализованы вашим производителем EJB. Эти интерфейсы, по существу, определяют услуги, предоставляемые контейнером компонентов, услуги, которые должны быть реализованы самим компонентом, и клиентский интерфейс компонента. Пакет `javax.ejb` также содержит ряд исключений, возбуждаемых компонентами.

EJB 2.0: AccessLocalException

Это стандартное системное исключение генерируется локальными компонентными интерфейсами для указания на то, что вызывающий объект (компонент) не имеет права вызывать определенный метод.

```
public class AccessLocalException extends EJBException
{
   public AccessLocalException();
```

```
    public AccessLocalException(String message);
    public AccessLocalException(String message, Exception ex);
}
```

CreateException

Это стандартное прикладное исключение должно возбуждаться всеми конструирующими методами, определенными во внутреннем интерфейсе. Оно указывает на то, что компонент не может быть создан.

```
public class javax.ejb.CreateException extends java.lang.Exception
{
    public CreateException();
    public CreateException(String message);
}
```

DuplicateKeyException

Это стандартное прикладное исключение возбуждается конструирующими методами внутреннего интерфейса объектных компонентов. Оно указывает на то, что компонент с таким первичным ключом уже существует.

```
public class javax.ejb.DuplicateKeyException extends
javax.ejb.CreateException
{
    public DuplicateKeyException();
    public DuplicateKeyException(String message);
}
```

EJBContext

Это базовый класс для классов `EntityContext`, `SessionContext` и `MessageDrivenContext`. `EJBContext` представляет собой средство для взаимодействия между классом компонента и контейнерной системой. Он предоставляет информацию об идентификаторах безопасности компонентов и состоянии транзакции. Кроме того, он предоставляет доступ к системным переменным и внутреннему объекту компонента.

```
public interface javax.ejb.EJBContext
{
    public abstract Principal getCallerPrincipal();
    public abstract EJBHome getEJBHome();
    public abstract EJBLocalHome getEJBLocalHome(); // новый в 2.0
    public abstract boolean getRollbackOnly();
    public abstract UserTransaction getUserTransaction();

    public abstract Properties getEnvironment(); // запрещен
    public abstract Identity getCallerIdentity(); // запрещен
    public abstract boolean isCallerInRole(Identity role); // запрещен
```

```
    public abstract boolean isCallerInRole(String roleName);
    public abstract void setRollbackOnly();
}
```

EJBException

Это исключение времени выполнения возбуждается классом компонента в его прикладных методах и методах обратного вызова для того, чтобы указать на возникновение непредвиденного исключения. Исключение вызывает откат текущей транзакции и уничтожение экземпляра компонента.

```
public class javax.ejb.EJBException extends java.lang.RuntimeException
{
    public EJBException();
    public EJBException(String message);
    public EJBException(Exception exception);

    public Exception getCausedByException();
}
```

EJBHome

Этот интерфейс должен расширяться удаленным внутренним интерфейсом компонента, создаваемым разработчиком классом, который определяет методы жизненного цикла компонента. В удаленном внутреннем интерфейсе определяются конструирующие, поисковые и внутренние методы компонента. Этот интерфейс реализуется внутренним объектом компонента.

```
public interface javax.ejb.EJBHome extends java.rmi.Remote
{
    public abstract HomeHandle getHomeHandle();
    public abstract EJBMetaData getEJBMetaData();
    public abstract void remove(Handle handle);
    public abstract void remove(Object primaryKey);
}
```

EJB 2.0: EJBLocalHome

Этот интерфейс должен расширяться локальным внутренним интерфейсом компонента, создаваемым разработчиком классом, который определяет методы жизненного цикла компонента. В локальном внутреннем интерфейсе определяются конструирующие, поисковые и внутренние методы компонента. Этот интерфейс реализуется внутренним объектом компонента.

```
public interface EJBLocalHome
{
    public void remove(java.lang.Object primaryKey)
```

```
        throws RemoveException,EJBException;
}
```

EJB 2.0: EJBLocalObject

Этот интерфейс определяет основные функциональные возможности, связанные с доступом к локальным компонентам. Он реализуется компонентным объектом. Разработчик должен предоставить локальный интерфейс для компонента, в котором определены его прикладные методы, и этот локальный интерфейс должен расширять интерфейс `EJBLocalObject`.

```
public interface EJBLocalObject
{
    public EJBLocalHome getEJBLocalHome() throws EJBException;
    public Object getPrimaryKey() throws EJBException;
    public boolean isIdentical(EJBLocalObject obj) throws EJBException;
    public void remove() throws RemoveException, EJBException;
}
```

EJBMetaData

Этот интерфейс реализуется производителем контейнера с целью предоставления сериализуемого класса, содержащего информацию, относящуюся к компоненту.

```
public interface javax.ejb.EJBMetaData
{
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
    public abstract Class getPrimaryKeyClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
    public abstract boolean isStatelessSession();
}
```

EJBObject

Этот интерфейс определяет основные функциональные возможности, связанные с доступом к удаленным компонентам. Он реализуется компонентным объектом. Разработчик должен предоставить удаленный интерфейс для компонента, который определяет прикладные методы компонента, а этот удаленный интерфейс должен расширить интерфейс `EJBObject`.

```
public interface javax.ejb.EJBObject extends java.rmi.Remote
{
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
```

```
    public abstract boolean isIdentical(EJBObject obj);
    public abstract void remove();
}
```

EnterpriseBean

Этот интерфейс расширяется интерфейсами EntityBean, SessionBean и MessageDrivenBean. Он выполняет общие функции.

```
public interface javax.ejb.EnterpriseBean extends java.io.Serializable {}
```

EntityBean

Этот интерфейс должен быть реализован классом объектного компонента. Он предоставляет набор информационных методов обратного вызова, уведомляющих экземпляр компонента о предстоящих или произошедших изменениях в его жизненном цикле.

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbLoad();
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void ejbStore();
    public abstract void setEntityContext(EntityContext ctx);
    public abstract void unsetEntityContext();
}
```

EntityContext

Этот интерфейс является специальным типом EJBContext, который предоставляет методы для получения ссылки на компонентный объект EntityBean и первичного ключа. EntityContext предоставляет экземпляру компонента средство взаимодействия с контейнером.

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext
{
    public abstract EJBObject getEJBObject();
    public abstract Object getPrimaryKey();
    public abstract EJBLocalObject getEJBLocalObject();  //new in EJB 2.0
}
```

FinderException

Это стандартное прикладное исключение генерируется поисковыми методами, определенными во внутреннем интерфейсе, для указания на сбой, произошедший во время выполнения поискового метода.

```
public class javax.ejb.FinderException extends java.lang.Exception
```

```
{
    public FinderException();
    public FinderException(String message);
}
```

Handle

Этот интерфейс предоставляет клиенту сериализуемый объект, который можно использовать для получения удаленной ссылки на заданный компонент.

```
public interface javax.ejb.Handle extends java.io.Serializable
{
    public abstract EJBObject getEJBObject();
}
```

HomeHandle

Этот интерфейс предоставляет клиенту сериализуемый объект, который можно использовать для получения удаленной ссылки на внутренний объект компонента.

```
public interface javax.ejb.HomeHandleextends java.io.Serializable
{
    public abstract EJBHome getEJBHome();
}
```

EJB 2.0: MessageDrivenBean

Этот интерфейс должен реализовываться классом компонента, управляемого сообщениями. Он предоставляет набор информационных методов обратного вызова, уведомляющих экземпляр компонента о предстоящих или произошедших изменениях в его жизненном цикле.

```
public interface MessageDrivenBean extends EnterpriseBean
{
    public void ejbRemove()throws EJBException;
    public void setMessageDrivenContext(MessageDrivenContext ctx)
        throws EJBException;
}
```

EJB 2.0: MessageDrivenContext

Этот интерфейс является подтипом `EJBContext`, который не предоставляет никаких новых методов. Все его методы определены в `EJBContext`. `MessageDrivenContext` предоставляет экземпляру компонента средство взаимодействия с контейнером.

```
public interface MessageDrivenContext extends EJBContext {}
```

NoSuchEntityException

Это исключение EJBException обычно генерируется методами ejbLoad() и ejbStore() класса компонента для указания на то, что данный объект не существует. Например, это исключение будет возбуждено, если объектный компонент с постоянством, реализуемым компонентом, пытается считать свое состояние (в методе ejbLoad()) из записи, которая была удалена из базы данных.

```
public class javax.ejb.NoSuchEntityException extends javax.ejb.EJBException
{
    public NoSuchEntityException();
    public NoSuchEntityException(String message);
    public NoSuchEntityException(Exception exception);
}
```

ObjectNotFoundException

Это стандартное прикладное исключение генерируется поисковыми методами, которые возвращают единственный компонентный объект. Оно указывает на то, что не может быть найден ни один компонент, соответствующий указанным критериям.

```
public class javax.ejb.ObjectNotFoundException extends
javax.ejb.FinderException
{
    public ObjectNotFoundException();
    public ObjectNotFoundException(String message);
}
```

RemoveException

Это стандартное прикладное исключение возбуждается в удаляющих методах для указания на проблемы, возникшие при удалении компонента.

```
public class javax.ejb.RemoveException extends java.lang.Exception
{
    public RemoveException();
    public RemoveException(String message);
}
```

SessionBean

Этот интерфейс должен быть реализован классом сеансового компонента. Он предоставляет набор информационных методов обратного вызова, уведомляющих экземпляр компонента о предстоящих или произошедших изменениях его жизненного цикла.

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void setSessionContext(SessionContext ctx);
}
```

SessionContext

Этот интерфейс является специальным типом `EJBContext`, который предоставляет методы для получения ссылки на компонентный объект `SessionBean`. `SessionContext` предоставляет экземпляру компонента средство взаимодействия с контейнером.

```
public interface javax.ejb.SessionContext extends javax.ejb.EJBContext
{
    public abstract EJBObject getEJBObject();
}
```

SessionSynchronization

Этот интерфейс предоставляет экземпляру сеансового компонента с состоянием дополнительные уведомления обратного вызова. Эти методы обратного вызова уведомляют компонент об изменении его текущего состояния по отношению к транзакции.

```
public interface javax.ejb.SessionSynchronization
{
    public abstract void afterBegin();
    public abstract void afterCompletion(boolean committed);
    public abstract void beforeCompletion();
}
```

EJB 2.0: TransactionRequiredLocalException

Это стандартное системное исключение генерируется локальными компонентными интерфейсами для того, чтобы указать, что вызов не связан ни с каким контекстом транзакции, тогда как вызываемый метод требует активной транзакции. Эта ситуация может произойти, когда клиент, не связанный ни с какой транзакцией, вызывает метод с атрибутом транзакций `Mandatory`.

```
public interface TransactionRequiredLocalException extends EJBException
{
    public TransactionRequiredLocalException();
    public TransactionRequiredLocalException(String message);
}
```

EJB 2.0: TransactionRolledbackLocalException

Это стандартное системное исключение генерируется тогда, когда в результате некоторого сбоя, произошедшего во время выполнения метода, был выполнен откат транзакции вызывающего объекта или она была помечена как подлежащая откату. Это исключение сообщает вызывающему объекту, что дальнейшая работа, выполненная внутри этой транзакции, будет бесполезной.

```
public interface TransactionRolledbackLocalException extends EJBException
{
    public TransactionRolledbackLocalException();
    public TransactionRolledbackLocalException(String message);
    public TransactionRolledbackLocalException(String message,Exception ex);
}
```

EJB 2.0: Пакет javax.jms

В этом пакете находятся классы и интерфейсы службы сообщений Java (Java Message Service, JMS). Интерфейс `MessageListener` этого пакета важен для EJB 2.0.

MessageListener

Этот интерфейс должен реализовываться компонентами, управляемыми сообщениями. Его метод `onMessage()` используется для доставки во время выполнения объектов `Message` компоненту, управляемому сообщениями.

```
public interface MessageListener
{
    public void onMessage(Message message);
}
```

EJB 2.0: Пакет javax.ejb.spi

При помощи этого пакета производители EJB реализуют контейнерные системы EJB. Он не используется разработчиками компонентов.

HandleDelegate

При передаче удаленной ссылки на `EJBObject` или `EJBHome` из одной контейнерной системы в другую контейнер EJB использует объект `HandleDelegate` для преобразования ссылки `EJBObject` или `EJBHome` из своей специфичной формы в не зависящую от производителя форму, которая может передаваться между серверами разных производителей. `HandleDelegate` представляет собой средство взаимодействия, невидимое раз-

работчику компонентов и предназначенное только для производителей EJB. Как разработчику компонента вам не следует беспокоиться о нем.

```
public interface HandleDelegate {
  public EJBHome readEJBHome(ObjectInputStream istream)
        throws throws java.io.IOException,java.lang.ClassNotFoundException;
  public EJBObject readEJBObject(ObjectInputStream istream)
        throws throws java.io.IOException,java.lang.ClassNotFoundException;
  public void writeEJBHome(EJBHome ejbHome, ObjectOutputStream ostream)
        throws throws java.io.IOException;
  public void writeEJBObject(EJBObject ejbObject, ObjectOutputStream ostream)
        throws throws java.io.IOException;
}
```

B

Диаграммы состояний и последовательностей

Это приложение содержит диаграммы состояний и последовательностей для всех типов компонентов, рассмотренных в этой книге: управляемые контейнером и управляемые компонентом объектные компоненты, сеансовые компоненты без состояния и с состоянием, и компоненты, управляемые сообщениями. Хотя в этих диаграммах используется стандартный универсальный язык моделирования (Unified Modeling Language, UML), для моделирования характеристик компонентов времени выполнения потребовались некоторые расширения. В диаграммах состояний, например, методы обратного вызова и операции создания экземпляров класса показаны в виде части события перехода.

В диаграммах последовательностей классы, предоставляемые контейнером, такие как сам контейнер, компонентный и внутренний объекты показаны как отдельные классы, но заключенные в один блок. Сообщения, посылаемые классами, находящимися в контейнерной системе, считаются передаваемыми из контейнерной системы как из целого, а не обязательно от конкретного предоставляемого контейнером класса. Это обобщение необходимо из-за того, что взаимодействия между контейнером и компонентом характеризуются этими классами, но отличаются в реализации у каждого производителя. Конкретный источник сообщения несущественен, если известно, что его послала контейнерная система.

Объектные компоненты

Диаграмма состояний жизненного цикла объектного компонента

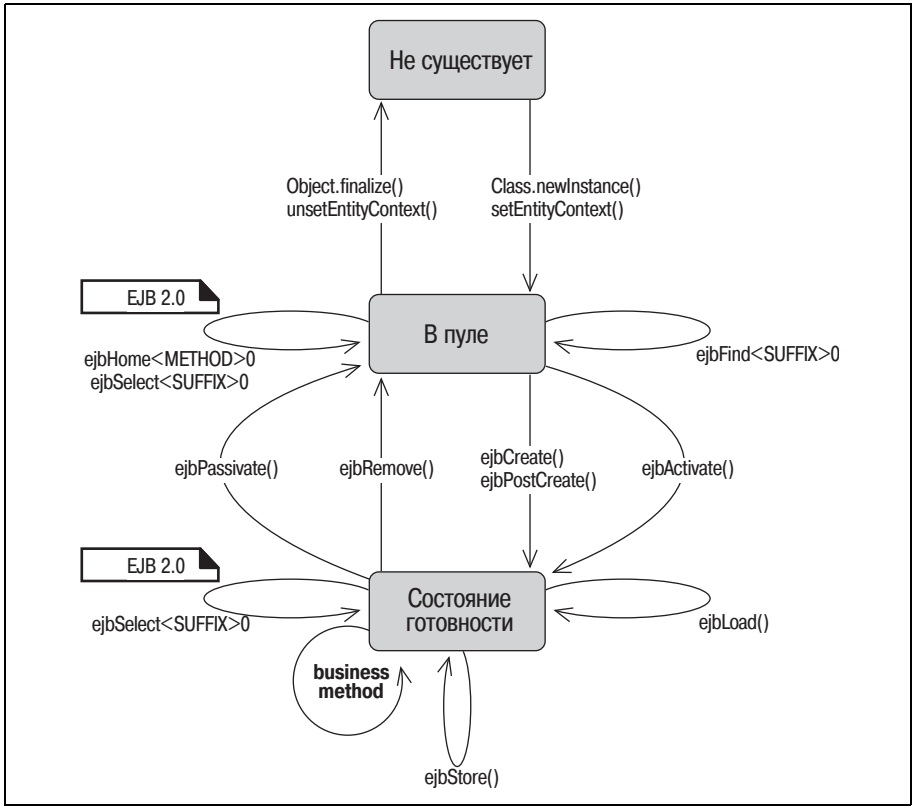

Рис. В.1. Диаграмма состояний жизненного цикла объектного компонента

Диаграммы последовательностей для постоянства, управляемого контейнером

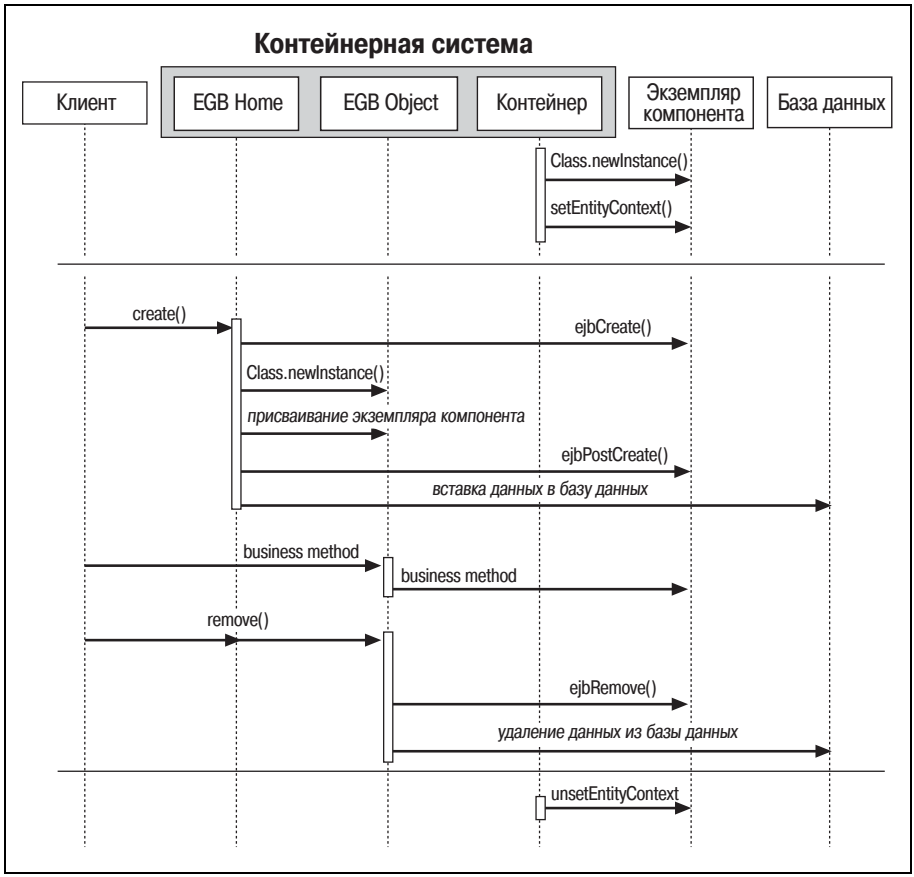

Рис. В.2. Процессы создания и удаления в постоянстве, управляемом контейнером

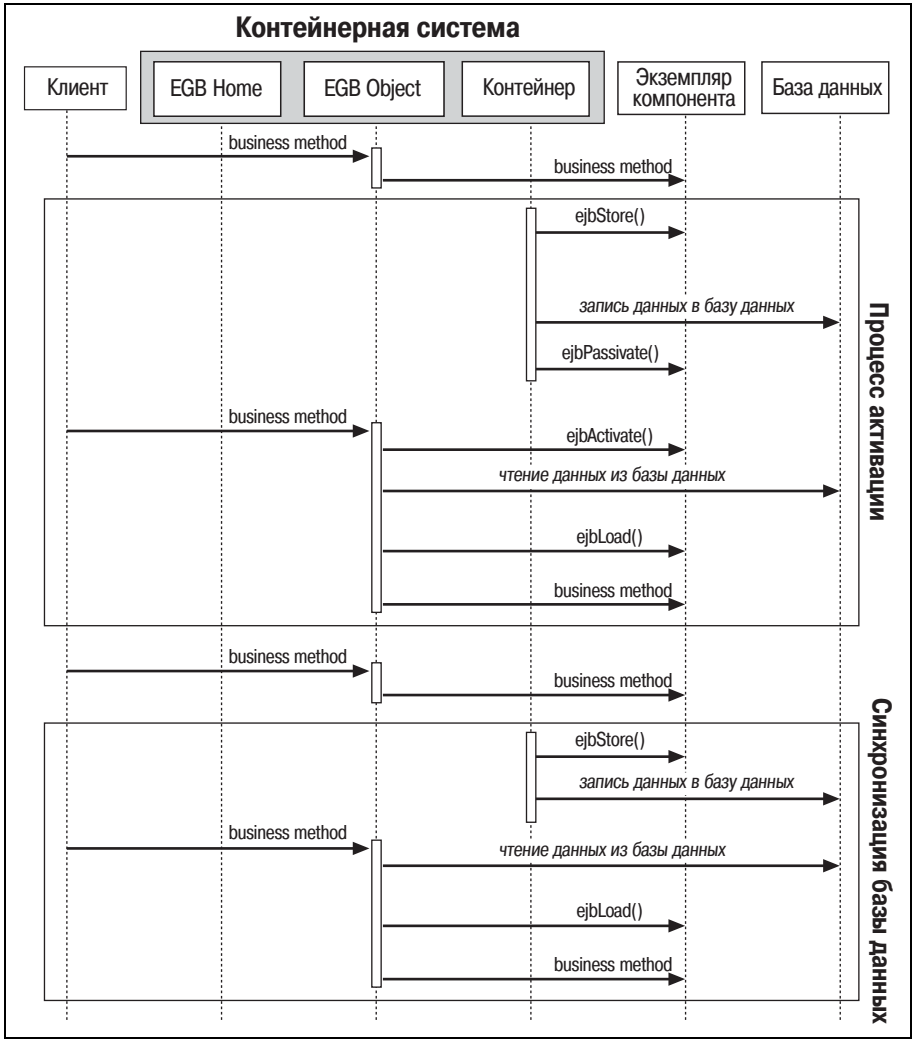

Рис. В.3. Процессы активации и синхронизации в постоянстве, управляемом контейнером

Диаграммы последовательностей для постоянства, управляемого компонентом

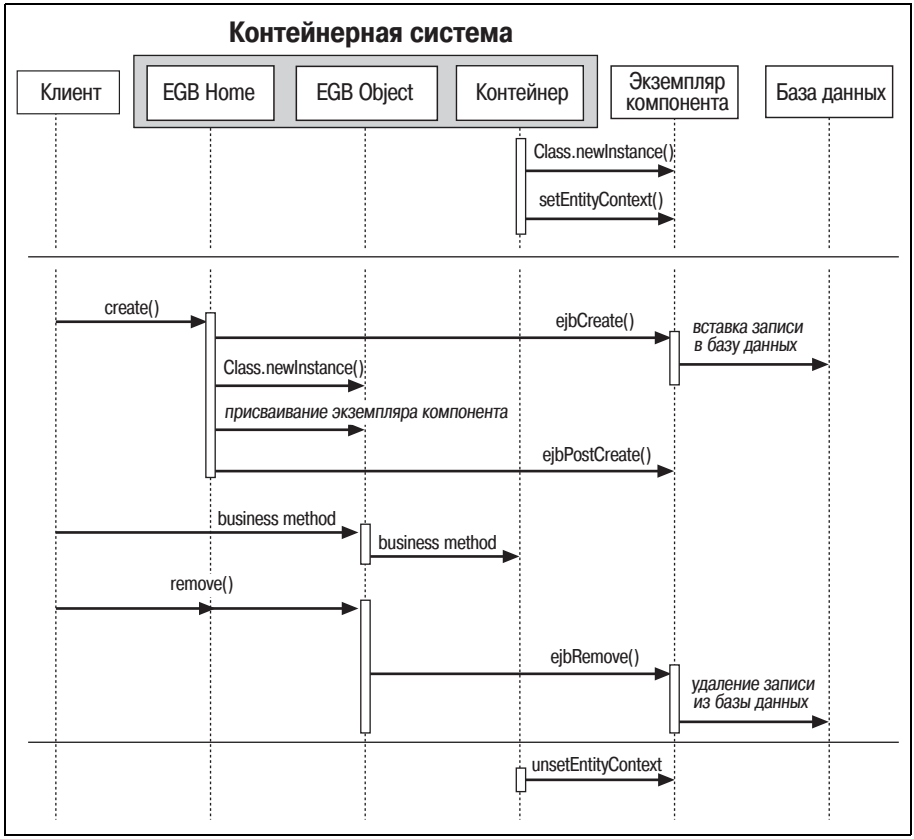

Рис. В.4. Процессы создания и удаления в постоянстве, управляемом компонентом

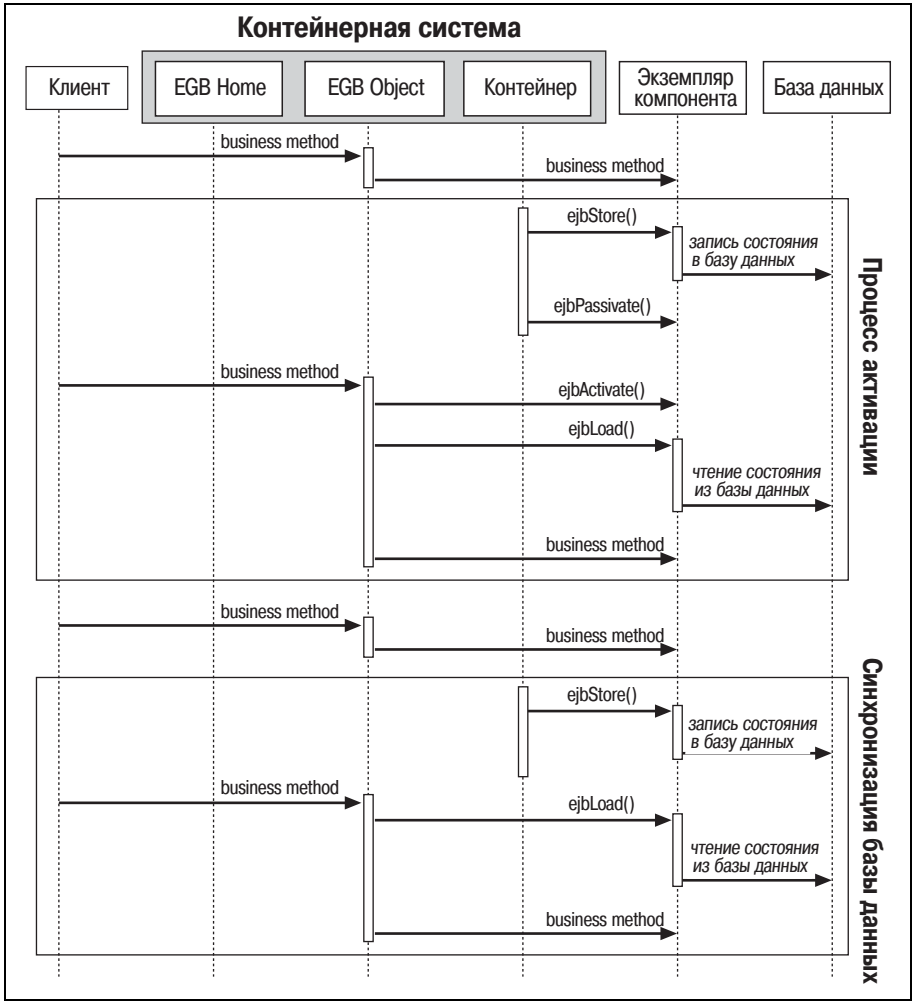

Рис. В.5. Процессы активации и синхронизации в постоянстве, управляемом компонентом

В табл. В.1 представлен обзор операций, которые разрешается выполнять компоненту на разных этапах своего жизненного цикла. Допустимые операции одинаковы для обеих версий EJB, за исключением методов `getEJBLocalHome()` и `getEJBLocalObject()` интерфейса `EntityContext` и методов `ejbHome()`, специфичных для EJB 2.0.

Заметьте, что объектные компоненты ни при каких условиях не могут вызывать метод `EJBContext.getUserTransaction()`, поскольку объектные компоненты не могут управлять своими собственными транзакциями. К этому методу могут обращаться лишь сеансовые компоненты.

Таблица В.1. Допустимые операции для объектных компонентов

Метод	Допустимые операции

`setEntityContext()`
`unsetEntityContext()`
	Методы `EntityContext`**:**
	`getEJBHome()`
	`getEJBLocalHome()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`

`ejbCreate()`
`ejbFind()`
`ejbHome()`
	Методы `EntityContext`**:**
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getRollbackOnly()`
	`setRollbackOnly()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов: `java:comp/env/jdbc`
	Компонентные ссылки: `java:comp/env/ejb`

`ejbPostCreate()`
`ejbLoad()`
`ejbStore()`
`ejbRemove()`
Прикладные методы
	Методы `EntityContext`**:**
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getRollbackOnly()`
	`setRollbackOnly()`
	`getEJBObject()`
	`getEJBLocalObject()`
	`getPrimaryKey()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов: `java:comp/env/jdbc`
	Компонентные ссылки: `java:comp/env/ejb`

`ejbActivate()`
`ejbPassivate()`
	Методы `EntityContext`**:**
	`getEJBHome()`
	`getEJBLocalHome()`
	`getEJBObject()`
	`getEJBLocalObject()`
	`getPrimaryKey()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`

Сеансовые компоненты

Сеансовые компоненты без состояния

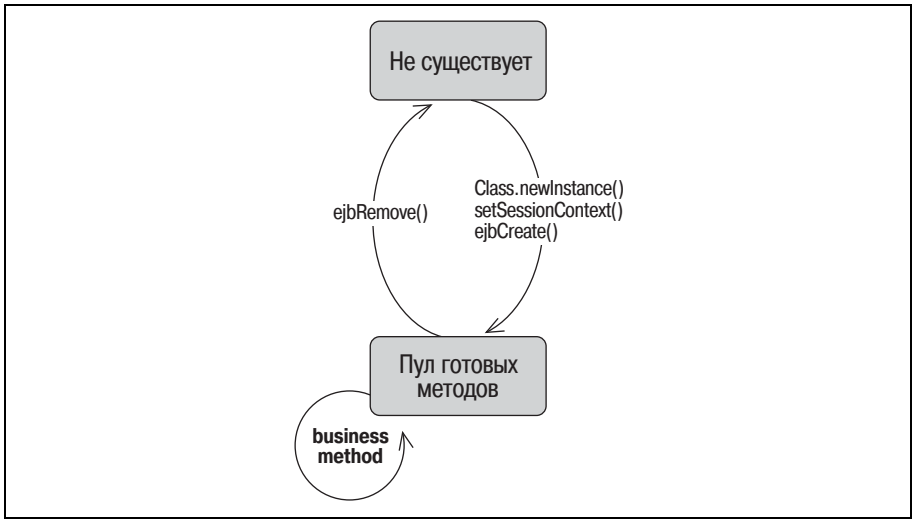

Рис. В.6. Диаграмма состояний жизненного цикла сеансового компонента без состояния

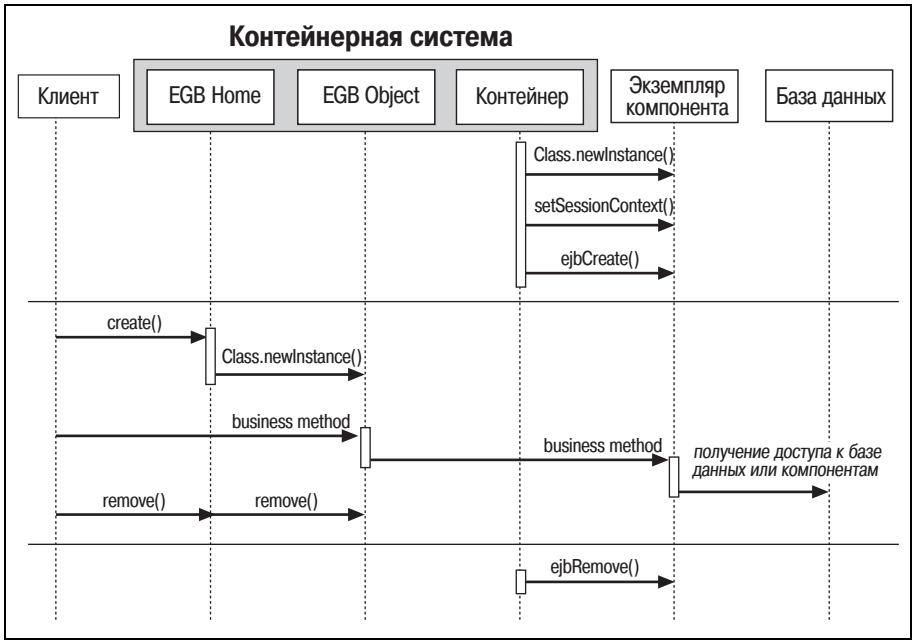

Рис. В.7. Процессы создания и удаления сеансового компонента без состояния

В табл. В.2 представлен обзор операций, разрешенных для сеансовых компонентов без состояния. Допустимые операции одинаковы для обеих версий EJB, за исключением методов `getEJBLocalHome()` и `getEJBLocalObject()` интерфейса `SessionContext`, специфичных для EJB 2.0.

Таблица В.2. Допустимые операции для сеансовых компонентов без состояния

	Допустимые операции	
Метод	**Транзакции, управляемые контейнером**	**Транзакции, управляемые компонентом**

`ejbCreate()`
`ejbRemove()`
	Методы `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
		Методы `EntityContext`:
		`getEJBHome()`
		`getEJBLocalHome()`
		`getEJBObject()`
		`getEJBLocalObject()`
		`getUserTransaction()`
		Контексты JNDI ENC:
		Свойства: `java:comp/env`

Прикладные методы	**Методы** `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getRollbackOnly()`
	`setRollbackOnly()`
	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов:
	`java:comp/env/jdbc`
	Компонентные ссылки:
	`java:comp/env/ejb`
		Методы `EntityContext`:
		`getEJBHome()`
		`getEJBLocalHome()`
		`getCallerPrincipal()`
		`isCallerInRole()`
		`getEJBObject()`
		`getEJBLocalObject()`
		`getUserTransaction()`
		Контексты JNDI ENC:
		Свойства: `java:comp/env`
		Менеджеры ресурсов:
		`java:comp/env/jdbc`
		Компонентные ссылки:
		`java:comp/env/ejb`

`ejbActivate()` `ejbPassivate()`	**Не поддерживаются (компоненты без состояния не поддерживают эти методы)**	**Не поддерживаются (компоненты без состояния не поддерживают эти методы)**

Сеансовые компоненты с состоянием

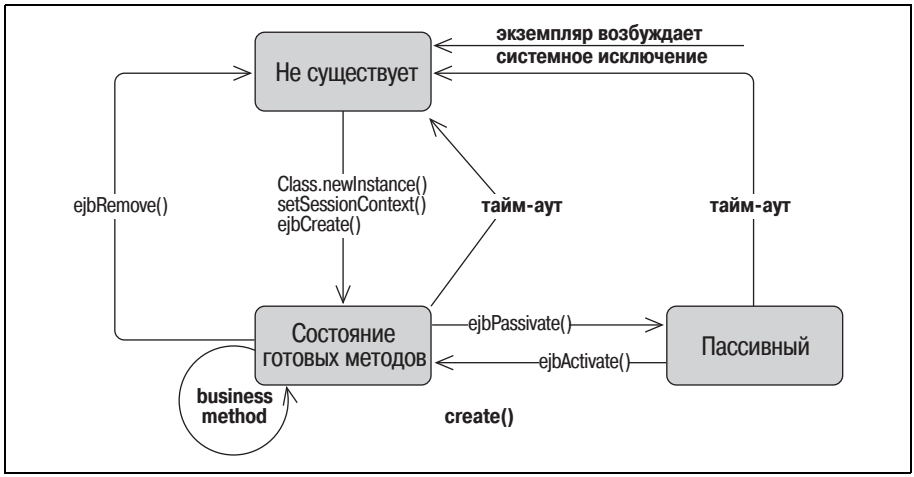

Рис. В.8. Диаграмма состояний жизненного цикла сеансового компонента с состоянием

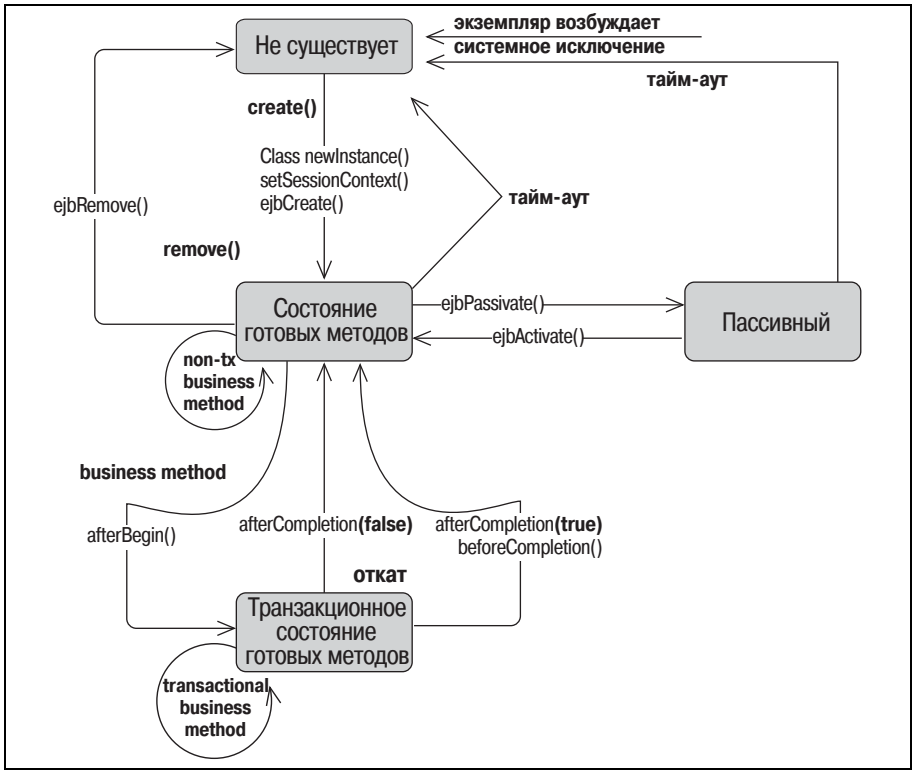

Рис. В.9. Жизненный цикл сеансового компонента с состоянием, использующего интерфейс SessionSynchronization

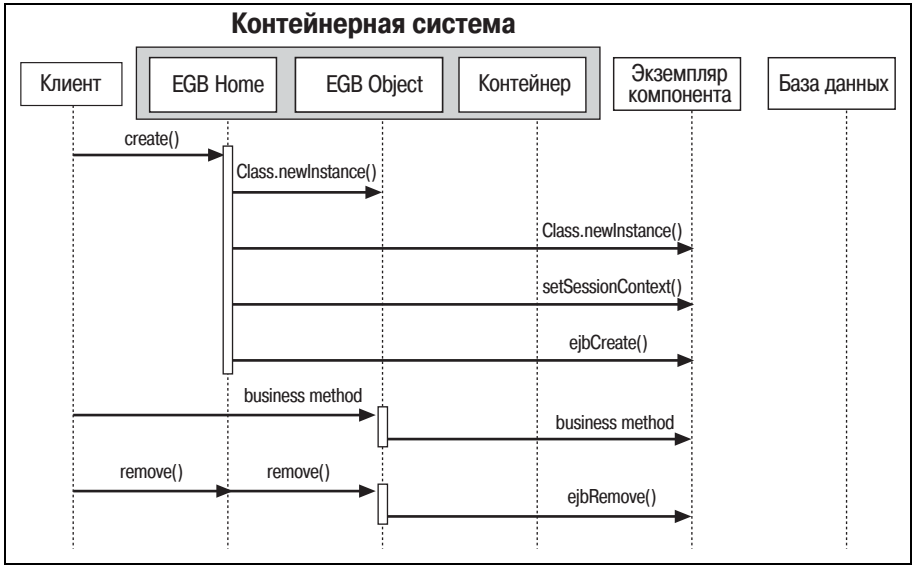

Рис. В.10. Процессы создания и удаления сеансовых компонентов с состоянием

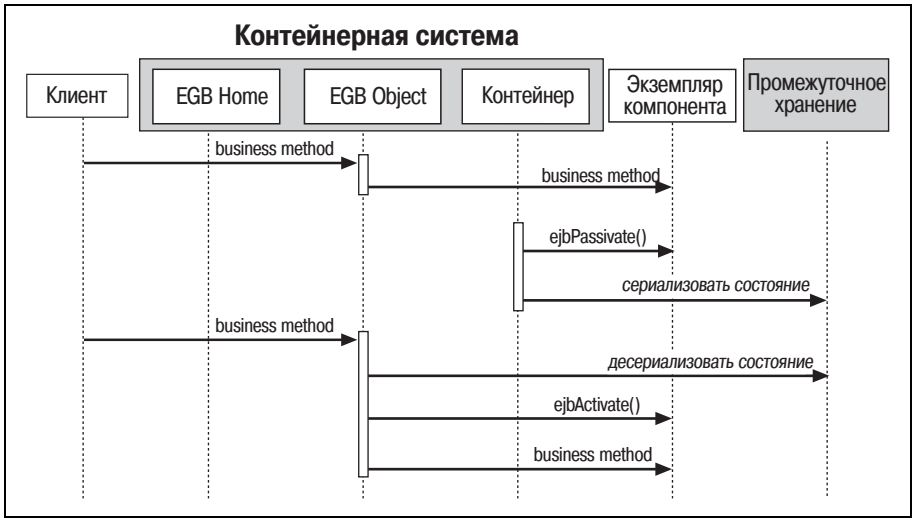

Рис. В.11. Процесс активации сеансовых компонентов с состоянием

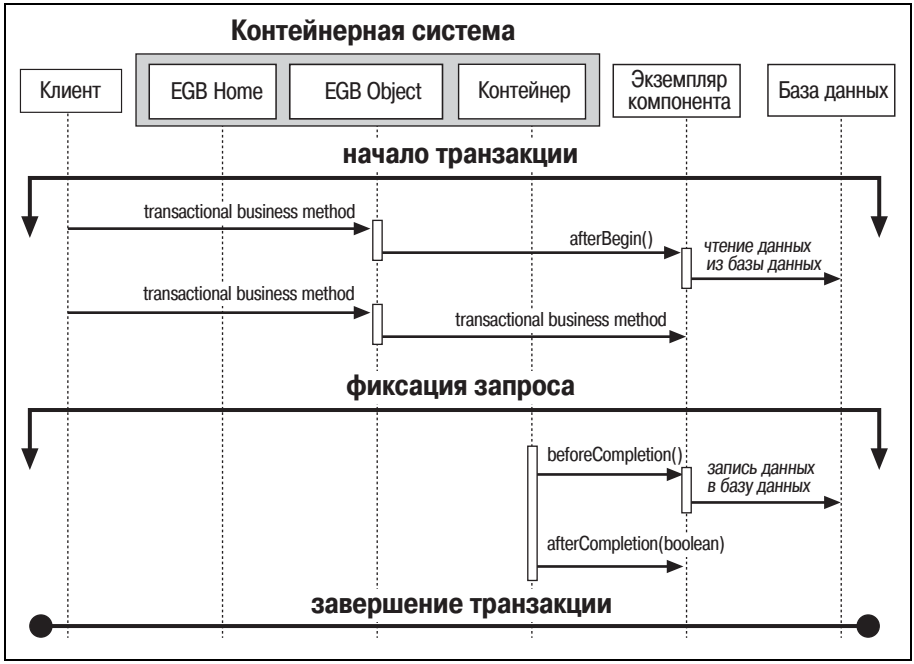

Рис. В.12. Транзакционные уведомления в сеансовых компонентах с интерфейсом SessionSynchronization.

В табл. В.3 представлен обзор операций, разрешенных для сеансовых компонентов с состоянием. Допустимые операции одинаковы для обеих версий EJB, за исключением методов `getEJBLocalHome()` и `getEJBLocalObject()` интерфейса `SessionContext`, специфичных для EJB 2.0.

Таблица В.3. Операции, допустимые для сеансовых компонентов с состоянием

	Допустимые операции	
Методы	**Транзакции, управляемые контейнером**	**Транзакции, управляемые компонентом**

`setSessionContext()`	**Методы** `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
		Методы `EntityContext`:
		`getEJBHome()`
		`getEJBLocalHome()`
		Контексты JNDI ENC:
		Свойства: `java:comp/env`

`ejbCreate()`
`ejbRemove()`
`ejbActivate()`
`ejbPassivate()`
	Методы `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
		Методы `EntityContext`:
		`getEJBHome()`
		`getEJBLocalHome()`
		`getCallerPrincipal()`
		`isCallerInRole()`

Таблица B.3 (продолжение)

Методы	Допустимые операции	
	Транзакции, управляемые контейнером	**Транзакции, управляемые компонентом**

	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов: `java:comp/env/jdbc`
	Компонентные ссылки: `java:comp/env/ejb`
		`getEJBObject()`
		`getEJBLocalObject()`
		`getUserTransaction()`
		Контексты JNDI ENC:
		Свойства: `java:comp/env`
		Менеджеры ресурсов: `java:comp/env/jdbc`
		Компонентные ссылки: `java:comp/env/ejb`

Прикладные методы	**Методы** `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getRollbackOnly()`
	`setRollbackOnly()`
	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов: `java:comp/env/jdbc`
	Компонентные ссылки: `java:comp/env/ejb`
		Методы `EntityContext`:
		`getEJBHome()`
		`getEJBLocalHome()`
		`getCallerPrincipal()`
		`isCallerInRole()`
		`getEJBObject()`
		`getEJBLocalObject()`
		`getUserTransaction()`
		Контексты JNDI ENC:
		Свойства: `java:comp/env`
		Менеджеры ресурсов: `java:comp/env/jdbc`
		Компонентные ссылки: `java:comp/env/ejb`

`afterBegin()`
`beforeCompetion()`
	Методы `EntityContext`:
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getRollbackOnly()`
	`setRollbackOnly()`
	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
	Менеджеры ресурсов: `java:comp/env/jdbc`
	Компонентные ссылки: `java:comp/env/ejb`
		Не поддерживаются (компоненты, управляющие транзакциями, не могут реализовывать интерфейс `SessionSynchronization`**)**

Таблица В.3 (продолжение)

Методы	Допустимые операции	
	Транзакции, управляемые контейнером	**Транзакции, управляемые компонентом**

`afterCompletion()`	**Методы** `EntityContext`**:**
	`getEJBHome()`
	`getEJBLocalHome()`
	`getCallerPrincipal()`
	`isCallerInRole()`
	`getEJBObject()`
	`getEJBLocalObject()`
	Контексты JNDI ENC:
	Свойства: `java:comp/env`
		Не поддерживаются (компоненты, управляющие транзакциями, не могут реализовывать интерфейс `SessionSynchronization`**)**

Компоненты, управляемые сообщениями

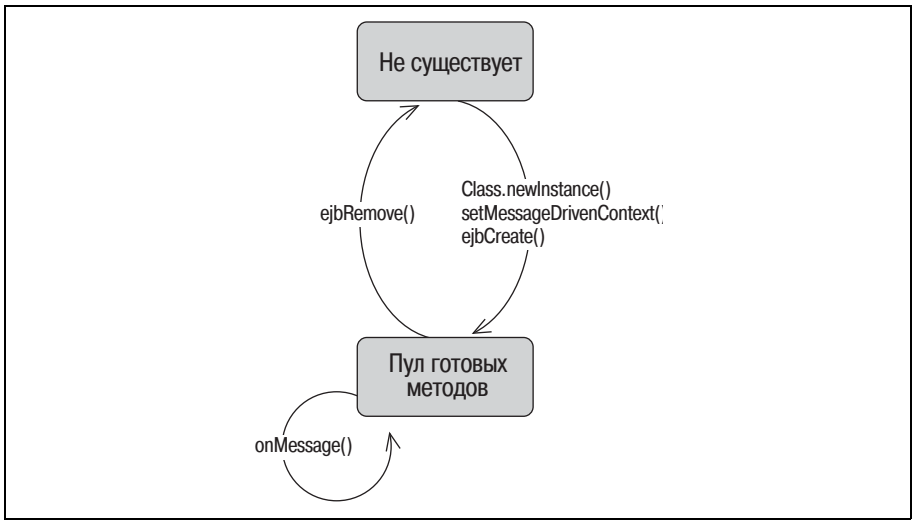

Рис. В.13. Диаграмма состояний жизненного цикла компонента, управляемого сообщениями

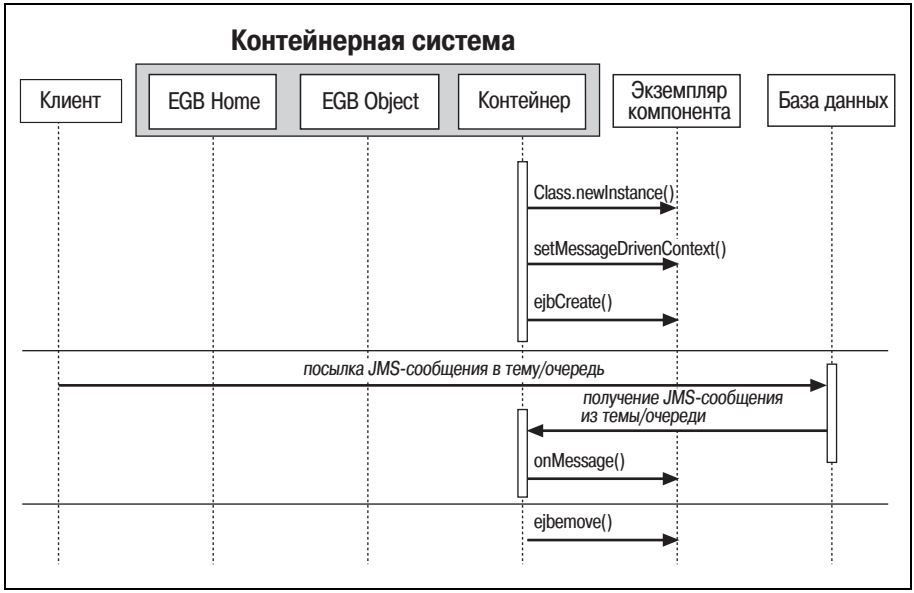

Рис. B.14. Диаграмма последовательностей компонента, управляемого сообщениями

Отношения между исключениями и транзакциями

В табл. B.4 показано, что происходит с транзакцией, если во время ее выполнения генерируется исключение.

Таблица B.4. Обзор исключений для сеансовых и объектных компонентов

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Клиентское представление
Транзакция, инициируемая клиентом. Транзакция начинается клиентом (приложением или компонентом) и распространяется на методы компонента.	Тип транзакции = Контейнерная Атрибуты транзакции = `Required` | `Mandatory` | `Support`	Прикладное исключение	Если вызван `setRollbackOnly()`, помечает транзакцию для отката. Повторно возбуждает прикладное исключение.	Принимает прикладное исключение. Транзакция клиента может быть помечена для отката.

Таблица В.4 (продолжение)

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Клиентское представление
		Системное исключение	Помечает транзакцию клиента для отката. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `TransactionRolledbackException` для удаленного клиента или `javax.TransactionRolledbackLocalException` для локальных в EJB 2.0.	Удаленные клиенты получают `TransactionRolledbackException`; локальные клиенты получают `javax.ejb.TransactionRolledbackException`. Выполняется откат транзакции клиента.
Транзакция, инициируемая контейнером. Транзакция начинается вызовом метода компонента и завершается по его окончанию.	Тип транзакции = Контейнерная Атрибут транзакции = `Require` | `RequiresNew`	Прикладное исключение	Если вызывается `setRollbackOnly()` компонента, выполняет откат транзакции и повторно возбуждает прикладное исключение. Если компонент не выполняет явный откат транзакции, пытается зафиксировать транзакцию и повторно возбудить прикладное исключение.	Получает прикладное исключение. Может быть выполнен откат транзакции компонента. Транзакция клиента не затрагивается.

Таблица В.4 (продолжение)

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Клиентское представление
		Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBException` – для локальных в EJB 2.0.	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBException`. Будет выполнен откат транзакции компонента. Транзакция клиента может быть помечена для отката в зависимости от производителя.
Компонент не является частью транзакции. Компонент вызывается, но не распространяет клиентскую транзакцию и не начинает свою.	Тип транзакции = Контейнерная Атрибут транзакции = `Never` | `NotSupported` | `Supports`	Прикладное исключение	Повторно возбуждает прикладное исключение.	Получает прикладное исключение. Клиентская транзакция не затрагивается.
		Системное исключение	Регистрирует ошибку. Удаляет компонент. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBException` для локальных клиентов в EJB 2.0.	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBException`. Клиентские транзакции могут быть помечены для отката в зависимости от производителя.

Таблица В.4 (продолжение)

Зона транзакции	Атрибуты типа транзакции	Возбуждаемые исключения	Действия контейнера	Клиентское представление
Транзакция, управляемая компонентом. Сеансовые компоненты с состоянием и без состояния используют `EJBContext` для явного управления своими транзакциями.	Тип транзакции = Компонентная Атрибут транзакции = Компоненты, управляющие транзакциями, не используют атрибуты транзакции.	Прикладное исключение	Повторно возбуждает прикладное исключение.	Получает прикладное исключение. Клиентская транзакция не затрагивается.
		Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет экземпляр. Повторно возбуждает `RemoteException` для удаленных клиентов и `EJBException` для локальных клиентов в EJB 2.0.	Удаленные клиенты получают `RemoteException`; локальные клиенты в EJB 2.0 получают `EJBException`. Клиентская транзакция не затрагивается.

В табл. В.5 представлен краткий обзор отношений между разными типами исключений и транзакций в компонентах, управляемых сообщениями.

Таблица В.5. Обзор исключений в компонентах, управляемых сообщениями

Зона транзакции	**Атрибуты транзакции**	**Возбуждаемое исключение**	**Действия контейнера**
Транзакция, инициируемая контейнером. Транзакция начинается перед вызовом метода `onMessage()` и заканчивается по его завершении.	Тип транзакции = Контейнерная Атрибут транзакции = `Required`	Системное исключение	Регистрирует ошибку. Удаляет экземпляр.
Транзакция, инициируемая компонентом.	Тип транзакции = Контейнерная Атрибут транзакции = `NotSupported`	Системное исключение	Регистрирует ошибку. Удаляет компонент.
Транзакция, управляемая компонентом. Компонент для явного управления своей транзакцией использует `EJBContext`	Тип транзакции = Компонентная Атрибут транзакции = компонент, управляющий транзакцией, не использует атрибуты транзакции.	Системное исключение	Выполняет откат транзакции. Регистрирует ошибку. Удаляет экземпляр.

C

Производители EJB

В этом приложении приведен список производителей серверов EJB. В него внесены все производители, которые мне известны по публикациям. Однако, учитывая, что число производителей, уже предоставляющих серверы, и тех, которые представляют новые продукты, ежедневно увеличивается, этот список, конечно же, неполон. Более того, я не пытался разграничить серверы EJB 2.0 и серверы 1.1. Большинство (если не все) производители EJB 1.1 должны перейти на 2.0 к середине 2002 года.

Коммерческие продукты

В табл. C.1 приведен список доступных на сегодняшний день коммерческих серверов EJB.

Таблица C.1. Коммерческие серверы EJB

Название компании	URL компании	Название сервера EJB
ATG	*http://www.atg.com*	Dynamo Application Server
BEA Systems	*http://www.beasys.com*	BEA WebLogic Server
Blazix	*http://www.blazix.com*	EJB Server
Borland	*http://www.borland.com*	AppServer
BROKAT Infosystems	*http://www.brokat.com*	Twister Gemstone/J

Название компании	URL компании	Название сервера EJB
Caucho Technology	*http://www.caucho.com*	Resin-CMP
Compaq	*http://nonstop.compaq.com*	NonStop™ Enterprise Application Server
Fujitsu-Seimens	*http://bs2www.fujitsu-siemens.de*	BeanTransactions
Fujitsu Software	*http://www.interstage.com*	Interstage
Hitachi	*http://www.hitachi.co.jp*	Cosminexus
HP Bluestone	*http://www.bluestone.com*	Total-e-Server
IBM	*http://www.software.ibm.com*	WebSphere Application Server
Information Builders	*http://www.ibi.com*	ParlayEJB Application Server
In-Q-My/SAP	*http://www.inqmy.com*	In-Q-My Application Server
Iona Technologies	*http://www.iona.com*	iPortal Application Server
ObjectSpace	*http://www.objectspace.com*	Voyager
Oracle	*http://www.oracle.com*	Oracle Application Server Oracle9i
Orion	*http://www.orionserver.com*	Orion Application Server
Persistence Software	*http://www.persistence.com*	PowerTier for J2EE
Pramati	*http://www.pramati.com*	Pramati Application Server
Secant Technologies	*http://www.secant.com*	ModelMethods Enterprise Server
Silverstream	*http://www.silverstream.com*	SilverStream Application Server
Sun Microsystems	*http://java.sun.com/j2ee/*	J2EE 1.3 SDK (эталонная реализация)
Sybase	*http://www.sybase.com*	EAServer
Unify	*http://www.unify.com*	EWave
Versant	*http://www.versant.com*	Versant enJin

Продукты с открытым кодом

С момента выхода первого издания этой книги стали доступными несколько серверов EJB с открытым исходным кодом. Они перечислены в табл. С.2. Автор этой книги, Ричард Монсон-Хейфел, выполняет обязанности ведущего архитектора сервера OpenEJB от ExoLab.

Таблица С.2. Серверы EJB с открытым кодом

Спонсор	URL компании	Название сервера EJB
Evidian/ Objectweb	*http://www.objectweb.org/jonas/*	JonAS
ExoLab	*http://openejb.exolab.org*	OpenEJB
JBoss	*http://www.jboss.org*	Jboss
Lutris	*http://www.enhydra.com*	Enhydra (сервер на базе JOnAS)

Алфавитный указатель

D

E

F

G

H

I

J

L

M

N

S

T

U

V

W

X

A

Б

В

Г

Д

K

Л

М

H

O

П

P

С

Т

У