ПРОФЕССИОНАЛЬНО

Диомидис Спинеллис
Георгиос Гусиос

ИДЕАЛЬНАЯ АРХИТЕКТУРА

ВЕДУЩИЕ СПЕЦИАЛИСТЫ О КРАСОТЕ ПРОГРАММНЫХ АРХИТЕКТУР

O'REILLY®

СИМВОЛ®

Beautiful Architecture

Leading Thinkers Reveal the Hidden Beauty in Software Design

Diomidis Spinellis and Georgios Gousios

O'REILLY®

ПРОФЕССИОНАЛЬНО

ИДЕАЛЬНАЯ АРХИТЕКТУРА

Ведущие специалисты о красоте программных архитектур

Диомидис Спинеллис и Георгиос Гусиос

СИМВОЛ®

Санкт-Петербург — Москва
2010

Серия «Профессионально»

Диомидис Спинеллис, Георгиос Гусиос

Идеальная архитектура

Ведущие специалисты о красоте программных архитектур

Перевод Е. Матвеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Шалин*
Научный редактор	*А. Брагин*
Редактор	*А. Альбов*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Спинеллис Д., Гусиос Г.

Идеальная архитектура. Ведущие специалисты о красоте программных архитектур. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 528 с., ил.

ISBN 978-5-93286-175-2

Из каких компонентов строятся надежные, элегантные, гибкие, удобные в сопровождении программные архитектуры? Книга отвечает на этот вопрос серией очерков, написанных ведущими программными архитекторами и проектировщиками современности. В каждом очерке авторы представляют какую-либо выдающую программную архитектуру, анализируют ее отличия от других архитектур и объясняют, почему она идеально подходит для своей цели.

Из книги вы узнаете, как на основе архитектуры Facebook была построена экосистема приложений, ориентированных на работу с данными; как новаторская архитектура Xen повлияла на будущее операционных систем; как процессы в сообществе проекта KDE способствовали превращению программной архитектуры из предварительного проекта в элегантную систему; как «ползучая функциональность» помогла GNU Emacs выйти за пределы изначально запланированных возможностей; как устроена высокооптимизированная виртуальная машина Jikes RVM; какие сходства и различия существуют между объектно-ориентированными и функциональными архитектурными школами; как архитектуры влияют на эволюцию программных продуктов и труд разработчиков.

ISBN 978-5-93286-175-2
ISBN 978-978-0-596-51798-4 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 21.05.2010. Формат 70x90 1/16 . Печать офсетная.
Объем 33 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Стивен Дж. Меллор

Проблемы разработки высокопроизводительных, надежных и качественных программных систем оказались слишком сложными для неформальных, несистематических технических решений, которые работали в прошлом в менее требовательных системах. Сложность наших систем дошла до того уровня, после которого с задачей невозможно справиться без разработки и поддержания общей архитектуры, которая объединяет систему в связное целое и предотвращает разрозненную реализацию со всеми вытекающими проблемами для тестирования и интеграции.

Однако построение архитектуры является весьма сложной задачей. Трудно найти хорошие примеры – иногда из-за закрытого характера решений, иногда, наоборот, из-за необходимости «протолкнуть» некоторый определенный архитектурный стиль в широком диапазоне сред, во многих из которых этот стиль неуместен. К тому же архитектуры весьма масштабны, поэтому их описания часто обескураживают читателей.

И все же в красивых архитектурах проявляются некоторые универсальные принципы, некоторые из которых я кратко опишу ниже:

Один факт в одном месте

Дублирование ведет к ошибкам, поэтому его необходимо избегать. Каждый факт должен быть единым и неделимым и притом независимым от всех остальных фактов. При внесении изменения (а это неизбежно происходит в любой системе) достаточно исправить только одно место. Этот принцип хорошо знаком проектировщикам баз данных, у которых он называется *нормализацией*. В менее формальной степени он также встречается в области проектирования

поведения, когда общая функциональность выделяется в отдельные модули (*факторинг*).

Красивые архитектуры находят способы локализации информации и поведения. Во время выполнения это проявляется в виде деления системы на уровни, каждый из которых представляет некоторый уровень абстракции или предметной области.

Автоматическое распространение

Один факт в одном месте – звучит, конечно, хорошо, но по соображениям эффективности некоторые данные или аспекты поведения часто дублируются. Для сохранения логической целостности и правильности системы распространение этих фактов должно происходить автоматически в ходе построения.

Красивые архитектуры имеют поддержку в виде строительного инструментария, реализующего принципы *метапрограммирования*. Другими словами, один факт в одном месте может распространяться во много мест для повышения эффективности его использования.

Архитектура включает построение

Архитектура должна включать не только систему времени выполнения, но и способ ее построения. Если архитектор уделяет внимание только коду времени выполнения, это верный признак того, что его архитектура постепенно придет в негодность.

Красивые архитектуры метацикличны. Они красивы не только на стадии выполнения, но и на стадии построения, в ходе которого используются те же данные, функции и приемы, что и во время выполнения.

Минимум механизмов

Лучший способ реализации некой функции зависит от конкретного случая, но красивая архитектура не стремится к «самому лучшему». Например, существует множество способов организации хранения данных и поиска в них, но если система может удовлетворить требования производительности, ограничившись использованием одного механизма, разработчику придется писать, проверять и сопровождать меньший объем кода.

Красивые архитектуры используют минимальный набор механизмов, удовлетворяющих общим требованиям. Поиск «самого лучшего» решения для каждого случая ведет к размножению механизмов с повышением риска ошибок, тогда как экономное добавление механизмов позволяет строить более компактные, быстрые и надежные системы.

Построение обобщенных решений

Если вы хотите строить хрупкие, негибкие системы, последуйте совету Айвара Якобсона (Ivar Jacobson) – заложите в основу своей архитектуры сценарии использования и реализацию только одной функции (т. е. используйте объекты-«контроллеры»).

Напротив, расширяемые системы базируются на создании виртуальных машин – обобщенных решений, которые «программируются» данными, предоставляемыми более высокими уровнями, и реализуют сразу несколько функций приложения.

Известно много разных формулировок этого принципа. «Уровни» виртуальных машин восходят еще к Эдгару Дейкстре. «Системы, управляемые данными» предоставляют механизмы, которые базируются на программных инвариантах системы и позволяют данным определить конкретную функциональность для конкретного случая. Они прекрасно подходят для повторного использования – и притом весьма красивы.

Порядок роста

Некогда мы говорили о «порядке» алгоритмов, выражая, скажем, сложность сортировки в виде времени, необходимого для сортировки определенного количества элементов. На эту тему были написаны целые книги.

То же относится и к архитектурным решениям. Например, опрос (polling) лучше всего работает для малого количества элементов, а с увеличением их количества время отклика катастрофически ухудшается. Построение архитектуры на базе прерываний или событий работает хорошо – до того момента, когда они вдруг срабатывают все сразу. Красивые архитектуры учитывают направление возможного роста и принимают соответствующие меры.

Сопротивление энтропии

Красивые архитектуры прокладывают путь наименьшего сопротивления для сопровождения, сохраняющего архитектуру и замедляющего действие закона системной энтропии, который гласит, что с течением времени система становится менее организованной. Программисты, занимающиеся сопровождением, должны хорошо понимать архитектуру, чтобы вносимые изменения соответствовали исходному замыслу, а не увеличивали энтропию системы.

Один из способов основан на применении понятия метафоры из гибких методологий – простого способа описания, «на что похожа» архитектура. Возможны и другие решения, например обширная документация и угрозы увольнения, хотя обычно это действует недолго.

> И все же, как правило, сопровождение основано на качественных инструментах, особенно для генерирования системы. Красивая архитектура должна оставаться красивой.

Эти принципы тесно связаны друг с другом. Принцип «один факт в одном месте» может работать только при наличии автоматического распространения, которое в свою очередь эффективно лишь тогда, когда архитектура учитывает процесс построения. Аналогичным образом конструирование обобщенных решений и минимизация механизмов способствуют соблюдению принципа «один факт в одном месте». Сопротивление энтропии обязательно для сопровождения архитектуры с течением времени, но оно зависит от того, были ли учтены в архитектуре процесс построения и поддержка распространения. Если не учесть направление наиболее вероятного роста системы, архитектура станет нестабильной и в конечном итоге развалится под воздействием экстремальных, но прогнозируемых обстоятельств. А объединение минимальности механизмов с концепцией конструирования обобщенных решений означает, что в красивых архитектурах обычно представлен ограниченный набор шаблонов, позволяющих строить произвольные расширения системы, своего рода «расширение по шаблону».

Короче говоря, красивые архитектуры делают больше меньшими средствами.

Во время чтения этой книги, которую так талантливо собрали и представили Диомидис Спинеллис и Георгиос Гусиос, обращайте внимание на эти принципы и рассматривайте их практическое применение на конкретных примерах, представленных в каждой главе. Также стоит присмотреться к нарушениям принципов и попытаться понять возможные причины – то ли архитектура некрасива, то ли действует какой-то принцип более высокого порядка.

Во время написания предисловия авторы спросили меня, не могу ли я сказать несколько слов о том, как стать хорошим архитектором. Я рассмеялся. Если бы мы знали, как... Но потом я вспомнил по собственному опыту, что существует достаточно мощный (хотя и не аналитический) способ стать хорошим архитектором. Он звучит так[1]: никогда не верьте, что последняя система, которую вы спроектировали, могла быть построена только так, а не иначе, ищите примеры различных решений аналогичных задач. Примеры красивых архитектур, представленные в книге, помогут вам достичь этой цели.

---

[1] Есть и другой вариант: больше тренироваться и меньше есть.

Вступление

Книга, которую вы сейчас читаете, зародилась в 2007 году как продолжение популярной, удостоенной наград книги «Beautiful Code»[1] – подборки статей о новаторских, часто удивительных решениях задач из области программирования. Данная книга отличается от предыдущей и масштабом, и предназначением, но общая идея осталась прежней: ведущие проектировщики и архитекторы описывают выбранную ими программную архитектуру, раскрывают особенности внутреннего строения своих творений и показывают, как они разрабатывают программы функциональные, надежные, удобные, эффективные, удобные в сопровождении и портировании... да, и элегантные.

За материалом для книги мы обратились к ведущим архитекторам хорошо известных или менее известных, но особенно оригинальных программных проектов. Многие из них быстро откликнулись на нашу просьбу и предложили свои идеи, заставляющие читателя задуматься. Некоторые даже застали нас врасплох, предложив вместо статьи о конкретной системе подробно проанализировать глубину и масштаб архитектурных аспектов программирования.

Все авторы были рады узнать, что их усилия в работе над книгой послужат доброму делу, так как авторские отчисления были пожертвованы «Врачам без границ» – международной гуманитарной организации, предоставляющей экстренную медицинскую помощь страдающим людям.

Структура книги

Материал разделен на пять тематических областей: обзоры, корпоративные приложения, системы, пользовательские приложения и язы-

[1] Энди Орам, Грег Уилсон «Идеальный код». – Пер. с англ. – СПб: Питер, 2008.

ки программирования. Бросается в глаза отсутствие глав, посвященных архитектурам программ для настольных систем, – не стоит полагать, что это было сделано преднамеренно. После обращения к более чем 50 проектировщикам программных продуктов такой результат стал для нас полной неожиданностью. Неужели в области программ для настольных систем нет действительно блестящих примеров красивых архитектур? Или талантливые архитекторы стараются держаться подальше от области, в которой развитие часто сводится к тому, что на приложение постоянно навешиваются все новые функции? Нам было бы интересно узнать ваше мнение по этому поводу.

Часть I «Об архитектуре»

В первой части книги изучается широта и масштаб программных архитектур, а также их значение для разработки и эволюции программ.

В главе 1 «Что такое архитектура?», написанной Джоном Клейном и Дэвидом Вайссом, программная архитектура определяется с точки зрения проблем качества и архитектурных структур.

Глава 2 «Повесть о двух системах: сказка для современных программистов», написанная Питом Гудлифом, в аллегорической форме показывает, как программные архитектуры могут влиять на эволюцию системы и на участие разработчиков в проектах.

Часть II «Архитектура корпоративных приложений»

Корпоративные системы – основа работы компьютерных служб многих организаций – часто представляют собой сделанные под заказ конгломераты, обычно построенные из разнообразных компонентов. Они обслуживают большие транзакционные нагрузки, а значит, должны масштабироваться вместе с потребностями предприятия, приспосабливаясь к изменяющимся экономическим реалиям. Важнейшие факторы при проектировании архитектуры таких систем – масштабируемость, корректность, стабильность и расширяемость. Во второй части книги представлены некоторые показательные примеры корпоративных программных архитектур.

Глава 3 «Масштабирование», написанная Джимом Уолдо, демонстрирует архитектурное мастерство, необходимое для построения серверов, обслуживающих массовые многопользовательские сетевые игры.

Глава 4 «Фото на память», написанная Майклом Найгардом, рассматривает архитектуру многоэтапной, многоузловой системы обработки данных, а также описывает компромиссы, необходимые для ее успешной работы.

Глава 5 «Ресурсно-ориентированные архитектуры, жизнь в WWW», написанная Брайаном Слеттеном, показывает возможности привязки к ресурсам при конструировании приложений, управляемых данными, и дает элегантный пример чистой ресурсно-ориентированной архитектуры.

Глава 6 «Архитектура платформы Facebook», написанная Дэйвом Феттерманом, разъясняет достоинства систем, в которых центральное место занимают данные, а также объясняет, как хорошая архитектура создает и поддерживает экосистему приложения.

Часть III «Системная архитектура»

Системные программы по праву считаются самыми сложными для проектирования – отчасти из-за того, что эффективная работа с оборудованием остается волшебством, доступным лишь немногим избранным, а отчасти из-за того, что многие относятся к системным программам как к инфраструктуре, которая «просто находится на своем месте». Великие системные архитектуры редко строятся на пустом месте; большинство современных систем базируется на идеях, впервые высказанных в 1960-е годы. В главах части III читатель познакомится с четырьмя новаторскими программными архитектурами, а также узнает, какие сложности, лежащие за архитектурными решениями, сделали их такими красивыми.

Глава 7 «Xen и красота виртуализации», написанная Дереком Мюрреем и Кайром Фрейзером, дает пример того, как хорошо продуманная архитектура изменяет пути эволюции операционных систем.

Глава 8 «Guardian: отказоустойчивая операционная система», написанная Грегом Лехи, содержит ретроспективный анализ архитектурных решений и структурных элементов (как программных, так и аппаратных), благодаря которым платформа Tandem почти 20 лет оставалась лидером в области сред высокой доступности.

Глава 9 «JPC: эмулятор x86 PC на языке Java», написанная Ризом Ньюманом и Кристофером Деннисом, описывает, как тщательное проектирование и хорошее понимание требований предметной области помогают преодолеть кажущиеся недостатки программной системы.

Глава 10 «Метациклические виртуальные машины: Jikes RVM», написанная Иэном Роджерсом и Дэйсом Гроувом, проведет читателя по пути принятия архитектурных решений, необходимых для создания самооптимизируемой, самодостаточной среды времени выполнения для высокоуровневого языка.

Часть IV «Архитектуры пользовательских приложений»

Пользовательские приложения – те программы, с которыми мы больше всего общаемся в своей повседневной жизни, и они же создают основную нагрузку на процессоры наших компьютеров. Такие программы обычно не нуждаются в тщательном управлении ресурсами или обслуживании больших объемов транзакций. Тем не менее пользовательские приложения должны быть удобными, безопасными, настраиваемыми и расширяемыми. Эти свойства делают их популярными и широко распространенными, а в случае бесплатных программ и программ с открытым кодом порождают армию добровольцев, желающих потрудиться над их совершенствованием. В части IV авторы анализируют архитектуры и процессы в сообществах, необходимые для эволюции двух очень популярных программных пакетов.

Глава 11 «GNU Emacs: сила ползучей функциональности», написанная Джимом Блэнди, объясняет, как набор очень простых компонентов и язык расширения превращают скромный текстовый редактор в ~~операционную систему~~[1] универсальный инструмент в рабочем арсенале программиста.

Глава 12 «Когда базар строит собор», написанная Тилем Адамом и Мирко Бёмом, показывает, как процессы в сообществах (такие как рабочие встречи и равноправный анализ кода) превращают программные архитектуры из предварительных набросков в прекрасные системы.

Часть V «Языки и архитектура»

Как сообщается во многих книгах по программированию, язык, который мы используем, влияет на способ решения задачи. Но может ли язык программирования также повлиять на архитектуру системы, и если да, то как? В строительной архитектуре новые материалы и распространение систем автоматизированного проектирования позволили выражать более сложные, а иногда невероятно красивые планы. Существует ли нечто аналогичное в области компьютерных программ? В части V, состоящей из двух последних глав, рассматриваются отношения между используемыми инструментами и создаваемыми архитектурами.

Глава 13 «Программные архитектуры: объектно-ориентированные и функциональные», написанная Бертраном Мейером, сравнивает пре-

[1] Как говорят некоторые фанатичные пользователи, «Emacs – моя операционная система, Linux просто предоставляет драйверы устройств».

имущества двух архитектурных стилей, объектно-ориентированного и функционального.

Глава 14 «Перечитывая классику», написанная Панайотисом Лурида-сом, содержит обзор архитектурных решений, заложенных в основу современных и классических объектно-ориентированных языков.

Наконец, в своем замечательном послесловии Уильям Дж. Митчелл, профессор архитектуры, мультимедийного искусства и науки Массачу-сетского технологического института, связывает концепции красоты зданий, возводимых в реальном мире, и программных архитектур, существующих только виртуально.

Принципы, свойства и структуры

На поздней стадии подготовки материалов один из рецензентов предложил высказать наше личное мнение, что читатель должен узнать из каждой главы. Идея была заманчивой, но нам не захотелось угадывать намерения авторов. Обратиться к самим авторам за метаанализом их материалов? Это кончилось бы возведением Вавилонской башни из определений, терминов и архитектурных конструкций, которые наверняка собьют с толку читателя. Нам была нужна общая номенклатура архитектурных терминов; к счастью, мы вовремя поняли, что она у нас уже есть.

В своем предисловии Стивен Мэллор обсуждает семь принципов, на которых базируются все красивые архитектуры. В главе 1 Джон Клейн и Дэвид Вайсс описывают четыре основных структурных элемента архитектур и шесть свойств, присущих красивым архитектурам. Внимательный читатель заметит, что принципы Мэллора и свойства Вайсса и Клейна частично взаимозависимы. Более того, они в основном совпадают; это происходит из-за того, что великие умы думают в похожих направлениях. Трое авторов – очень опытные архитекторы, и они неоднократно убеждались в важности описываемых ими принципов на собственном опыте.

Мы объединили архитектурные принципы Мэллора с определениями Клейна и Вайсса в двух списках: в первом перечислены принципы и свойства (табл. В.1), а во втором – структурные элементы (табл. В.2). Затем мы попросили авторов каждой статьи отметить те пункты, которые, по их мнению, относились к их материалу, и снабдили каждую главу соответствующим пояснением. В таблицах приведены определения всех принципов, свойств и архитектурных концепций, присутствующих в начале каждой главы. Надеемся, эти пояснения, дающие четкий обзор содержимого каждой главы, помогут вам ориентироваться в книге.

Таблица В.1. Архитектурные принципы и свойства

Принцип или свойство	Означает, что архитектура...
Гибкость	Предоставляет «подходящие» механизмы для решения разнообразных задач с относительно небольшим объемом выразительных средств.
Концептуальная целостность	Предоставляет один оптимальный, лишенный избыточности способ выражения решения группы сходных задач.
Возможность независимого изменения	Имеет изолированные элементы, что сводит к минимуму количество мест внесения изменений при модификации.
Автоматическое распространение	Поддерживает логическую целостность и правильность работы посредством распространения изменений данных или поведения между модулями.
Удобство построения	Управляет правильным и логичным процессом построения программного продукта.
Адаптация к росту	Может приспособиться к возможному росту.
Сопротивление энтропии	Поддерживает порядок за счет принятия, ограничения и изоляции последствий изменений.

Таблица В.2. Архитектурные структуры

Структура	Назначение
Модуль	Скрывает решения проектирования или реализации за стабильным интерфейсом.
Зависимость	Упорядочивает компоненты в соответствии с использованием той или иной функциональности.
Процесс	Инкапсулирует и изолирует состояние модуля на стадии выполнения.
Доступ к данным	Разбивает данные на категории с назначением прав доступа к ним.

Условные обозначения

В книге используются следующие условные обозначения:

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`

Используется в листингах программ, а также в тексте для обозначения программных элементов (имена переменных и функций, баз данных, типов данных, переменных среды, команд и ключевых слов).

`Моноширинный полужирный шрифт`

Команды или другой текст, который должен вводиться в показанном виде.

`Моноширинный курсив`

Текст, который должен заменяться пользовательскими значениями (или значениями, определяемыми по контексту).

Использование примеров кода

Эта книга призвана помочь вам в решении конкретных задач. В общем случае вы можете использовать приводимые примеры кода в своих программах и документации. Связываться с авторами для получения разрешения не нужно, если только вы не воспроизводите значительный объем кода. Например, если ваша программа использует несколько фрагментов кода из книги, обращаться за разрешением не нужно. С другой стороны, для продажи или распространения дисков CD-ROM с примерами из книг O'Reilly потребуется разрешение. Если вы отвечаете на вопрос на форуме, приводя цитату из книги с примерами кода, обращаться за разрешением не нужно. Если значительный объем кода из примеров книги включается в документацию по вашему продукту, разрешение необходимо.

Мы будем признательны за ссылку на источник информации, хотя и не требуем ее. Обычно в ссылке указывается название, автор, издательство и код ISBN, например: «*Beautiful Architecture*, edited by Diomidis Spinellis and Georgios Gousios. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51798-4».

Если вы полагаете, что ваши потребности выходят за рамки оправданного использования примеров кода или разрешений, приведенных выше, свяжитесь с нами по адресу *permissions@oreilly.com*.

Safari® Enabled

Если на обложке вашей любимой технической книги есть пиктограмма Safari® Enabled, это означает, что книга доступна через O'Reilly Network Safari Bookshelf.

Система Safari лучше обычных электронных книг. Это целая виртуальная библиотека с возможностью поиска по тысячам лучших технических книг, копирования/вставки примеров кода, загрузки глав и быстрого поиска ответов, когда вам потребуется самая точная и актуальная информация. Safari можно бесплатно опробовать на сайте *http://safari.oreilly.com*.

Как с нами связаться

Пожалуйста, со всеми комментариями и вопросами по книге обращайтесь к издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (США или Канада)
707-829-0515 (международные или местные звонки)
707 829-0104 (факс)

На сайте издательства имеется веб-страница книги со списками обнаруженных опечаток и другой дополнительной информацией. Страница доступна по адресу:

http://www.oreilly.com/catalog/9780596517984

С комментариями и техническими вопросами по поводу книги обращайтесь по электронной почте:

bookquestions@oreilly.com

За дополнительной информацией о книгах, конференциях, ресурсах и O'Reilly Network обращайтесь на сайт O'Reilly:

http://www.oreilly.com

Благодарности

Издание книги является делом коллективным, особенно если эта книга – сборник статей. Мы благодарны многим людям. Прежде всего, мы благодарим соавторов, которые своевременно предоставили материал, а потом внимательно относились к нашим просьбам по поводу различных изменений и дополнений. Рецензенты книги, Роберт А. Максимчук (Robert A. Maksimchuk), Гэри Поллис (Gary Pollice), Дэвид Уэст (David West), Грег Уилсон (Greg Wilson) и Бобби Янг (Bobbi Young), поделились полезными замечаниями по поводу улучшения каждой главы и книги в целом. Наш редактор в издательстве O'Reilly Мэри Треслер (Mary Treseler) помогла нам с поиском авторов, организовала

процесс рецензирования и управляла ходом работы с потрясающей эффективностью. Позднее Сара Шнейдер (Sarah Schneider) работала с нами в качестве редактора по производству, в сжатые сроки умело справляясь с нередко противоречивыми требованиями. Выпускающий редактор Женевьева д'Антремон (Genevieve d'Entremont) и составитель алфавитного указателя Фред Браун (Fred Brown) мастерски сформировали из материала, собранного со всего мира, книгу, которая читается легко, словно написанная одним пером. Художник Роберт Романо (Robert Romano) преобразовал графику в разных форматах (включая наброски, сделанные от руки) в профессиональные диаграммы, которые вы видите в книге. Дизайнер обложки Карен Монтгомери (Karen Montgomery) создала красивую и привлекательную обложку, соответствующую содержанию книги, а художник-оформитель Дэвид Футато (David Futato) предложил оригинальную и удобную схему интеграции сводки пояснений в дизайн книги. Наконец, мы хотим поблагодарить за поддержку свои семьи и друзей, ведь мы уделяли этой книге то внимание, которое по праву принадлежало им.

I

Об архитектуре

1

Что такое архитектура?

Джон Клейн
Дэвид Вайсс

Введение

Термин «архитектура» используют строители, музыканты, писатели, проектировщики компьютеров и сетей, программисты (и не только они; когда-нибудь слышали о «кулинарных архитекторах»?) – но с разными результатами. Здание совершенно не похоже на симфонию, но у того и другого есть архитектура. Кроме того, все архитекторы непременно упоминают о красоте своей работы и ее результатов. Строитель скажет, что здание должно создавать хорошие условия для жизни или работы и что оно должно быть красивым. Музыкант скажет, что музыка должна быть хорошо исполнена, в ней должна четко прослеживаться тема и она должна быть красивой. Программный архитектор скажет, что система должна быть дружественной к пользователю и быстро реагировать на его действия, быть простой в установке и сопровождении, надежной и свободной от критических ошибок; кроме того, система должна стандартно взаимодействовать с другими системами и – тоже должна быть красивой.

В этой книге приводятся подробно описанные примеры красивых архитектур из области компьютеризированных систем. Поскольку наша

дисциплина относительно молода, у нас меньше образцов для подражания, чем в таких областях, как строительство, музыка или литература, а потому потребность в них еще сильнее. Книга призвана заполнить этот пробел.

Прежде чем переходить к конкретным примерам, нам хотелось бы разобраться, что же такое архитектура вообще и какими признаками обладает красивая архитектура. Как видно из разных определений архитектуры, приводимых в этой главе, в каждой области существует собственное определение, поэтому мы сначала исследуем общие признаки архитектур из разных областей и проблем, решаемых с их помощью. В частности, архитектура помогает нам обеспечить соответствие системы потребностям заинтересованных сторон, а также справиться со сложностями планирования, построения и сопровождения системы.

Далее мы переходим к определению архитектуры. Вы увидите, как это определение применяется к архитектуре программных продуктов, поскольку именно они занимают центральное место во многих последующих примерах. Ключевым моментом определения является то, что архитектура состоит из набора структур, спроектированных специально для того, чтобы архитекторы, строители и другие заинтересованные стороны могли видеть, в какой степени система удовлетворяет их потребностям.

Глава завершается обсуждением признаков красивых архитектур; мы рассмотрим несколько примеров. В красоте центральное место занимает концептуальная целостность – набор абстракций и правил их по возможности простого использования в системе.

В нашем обсуждении термином «архитектура» будет обозначаться совокупность артефактов (включая документацию: планы, строительные спецификации и т. д.), описывающих создаваемый объект, при этом сам объект рассматривается как набор структур. Некоторые авторы также используют термин для описания процесса создания артефактов, включая конечный результат работы. Но, как указывают Джим Уолдо (Jim Waldo) и другие, не существует методологии, которая гарантировала бы создание хорошей архитектуры, не говоря уже о красивой (Waldo 2006), поэтому наше внимание будет в большей степени сосредоточено на артефактах, а не на процессах.

Архитектура: «Искусство или наука строительства; прежде всего, искусство и практика проектирования и строительства зданий, предназначенных для использования человеком, с учетом эстетических и практических факторов».

Краткий Оксфордский словарь английского языка, 5-е издание, 2002

Во всех отраслях знаний архитектура предоставляет средства для решения стандартной задачи: обеспечения того, что построенное здание, мост, музыкальное сочинение, книга, компьютер, сеть или система будут обладать определенными свойствами или функциями. Иначе говоря, архитектура является как планом, который гарантирует наличие у системы желаемых свойств, так и описанием построенной системы. В Википедии говорится: «Согласно самой ранней из дошедших до нас работ по теме – трактату Витрувия «Об архитектуре» – хорошее здание должно обладать красотой (Venustas), прочностью (Firmitas) и практичностью (Utilitas); архитектура может быть определена как баланс и сочетание этих трех элементов, при котором ни один не подчиняет себе остальные».

Мы говорим об «архитектуре» симфонии,
и называем архитектуру «застывшей музыкой».

Дерик Кук «The Language of Music»

Хорошей архитектуре системы присуща концептуальная целостность; другими словами, она дополняется набором правил проектирования, которые способствуют снижению сложности системы и могут использоваться как руководство по ее проектированию и проверке. Правила проектирования могут включать некоторые абстракции, которые всегда используются одинаково, например виртуальные устройства. Правила также могут быть представлены в виде моделей, или шаблонов, скажем каналов и фильтров. В лучшем случае архитектура включает правила, поддающиеся объективной проверке, например: «любое виртуальное устройство определенного типа может заменить любое другое виртуальное устройство того же типа в случае его отказа» или «все процессы, конкурирующие за один ресурс, должны обладать одинаковым приоритетом планирования».

Современный архитектор может сказать, что конструируемый объект или система должны обладать следующими характеристиками:

- Функциональность, необходимая заказчику
- Возможность построения в заданный срок
- Адекватность функционирования
- Надежность
- Удобство использования
- Безопасность
- Приемлемость по затратам
- Соответствие законодательству
- Превосходство в сравнении с предшественниками и конкурентами

Архитектура компьютерной системы – это минимальный набор свойств, определяющих, какие программы будут работать в системе и какие результаты они дадут.

Джеррит Блау и Фредерик Брукс «Computer Architecture»

Мы еще не видели ни одной сложной системы, которая бы идеально соответствовала всем перечисленным характеристикам. Архитектура является искусством компромисса – решение, усиливающее одну из этих характеристик, часто ослабляет другие. Архитектор должен определить, в какой степени достаточно удовлетворять те или иные требования, для чего он выявляет важные запросы конкретной системы и приемлемые уровни их выполнения.

Одно из основных понятий архитектуры – концепция структур, определяемых различными компонентами и их отношениями: согласованиями, коммуникациями, синхронизациями и другими взаимодействиями. Такими компонентами могут быть опорные балки или внутренние помещения в здании, отдельные партии музыкальных инструментов или мелодии в симфонии, главы или персонажи книги, процессоры и микросхемы памяти в компьютере, уровни стека коммуникационных протоколов или процессоры, подключенные к сети, взаимодействующие последовательные процессы, объекты, коллекции макросов времени компиляции или сценарии времени выполнения. В каждой отрасли знаний существуют свои виды компонентов и отношений между ними.

В более широком понимании термин «архитектура» всегда означает «неизменная глубинная структура».

Стюарт Бранд «How Buildings Learn»

Несмотря на все возрастающую сложность систем и их взаимодействий (как внутренних, так и внешних), архитектура, складывающаяся из набора структур, предоставляет основные средства для снижения уровня сложности- и обеспечения необходимых свойств у итоговой системы. Структуры дают возможность интерпретировать систему как набор взаимодействующих компонентов.

Каждая структура призвана помочь архитектору понять, каким образом удовлетворить ту или иную потребность, например способность к изменениям или производительность системы. Возможно, доказывать, что конкретная потребность действительно удовлетворяется, придется кому-то другому, но архитектор должен быть способен продемонстрировать, что *все* потребности были учтены при проектировании.

Сетевая архитектура: коммуникационное оборудование, протоколы и каналы передачи данных, образующие сеть, и методы их организации.

http://www.wtcs.org/snmp4tpc/jton.htm

Роль архитектора

При проектировании, строительстве или реконструкции зданий ведущие проектировщики назначаются «архитекторами», и на них возлагается широкий круг обязанностей. Архитектор готовит исходные эскизы, на которых обозначены внешний вид здания и внутренняя структура помещений, затем обсуждает их с клиентами, пока все заинтересованные стороны не согласятся с тем, что изображено именно то, что им нужно. Эскизы являются абстракциями: основное внимание на них уделяется наиболее существенным подробностям конкретного аспекта здания, а другие аспекты игнорируются.

После того как архитектор согласует эти абстракции со своими клиентами, он готовит гораздо более подробные чертежи (или наблюдает за их подготовкой), а также соответствующие текстовые спецификации. Эти чертежи со спецификациями описывают многие «житейские подробности» здания: трубопроводы, материал наружной отделки, остекление окон и электропроводку.

Изредка архитектор просто передает подробные чертежи строителю, который возводит по ним здание. В более важных проектах архитектор продолжает участвовать в строительстве, периодически проверяет работу, а также может предлагать изменения или принимать пожелания как от строителя, так и от клиента. Если архитектор наблюдает за ходом проекта, то последний считается законченным лишь после того, как архитектор подтвердит, что построенное здание в основном соответствует всем чертежам и спецификациям.

Мы привлекаем к работе архитектора, чтобы убедиться в том, что проект: (1) удовлетворяет потребностям клиента, включая упоминавшиеся ранее характеристики; (2) обладает концептуальной целостностью, то есть во всех аспектах проекта используются одни и те же правила проектирования; (3) удовлетворяет существующим законам и требованиям безопасности. Одна из важных обязанностей архитектора заключается в наблюдении за последовательным применением концепций проектирования в ходе реализации проекта.

Иногда архитектор также выполняет функции посредника между строителем и клиентом. По поводу того, какие решения находятся в ведении архитектора, а какие принимаются другими людьми, часто возникают разногласия, но всегда ясно, что архитектор принимает все критически важные решения, в том числе те, которые могут повлиять

на удобство использования, безопасность и эксплуатационную надежность конструкции.

Сочинение музыки и программные архитектуры

Хотя разработку программных архитектур часто сравнивают со строительством зданий, возможно, сочинение музыки является более правильной аналогией. Архитектор-строитель создает статическое описание (чертежи и другие рисунки) относительно статичной конструкции (относительно – потому что архитектура должна учитывать движение людей и вспомогательных механизмов в здании, а также распределение нагрузок). При сочинении музыки (и проектировании программных продуктов) композитор (архитектор программного продукта) создает статическое описание музыкального произведения (описание архитектуры и кода), которое позднее многократно исполняется (запускается). И в музыке, и в программировании архитектура может учитывать взаимодействие многочисленных компонентов для получения желаемого результата, а сам результат зависит от исполнителя, среды исполнения и интерпретации исходного замысла исполнителем.

Роль программного архитектора

В проектах по разработке программного обеспечения необходимы люди, которые играют в построении продукта ту же роль, что и традиционные архитекторы при строительстве или реконструкции зданий. Но в программных системах никогда не существовало полной ясности по поводу того, какие решения являются прерогативой архитектора, а какие можно оставить людям, занимающимся реализацией. Сложности с определением функций архитектора в программном проекте обусловлены тремя факторами: отсутствием традиций, нематериальной природой продукта и сложностью системы. (Панорама выполнения архитектором своих функций в крупной организации, занимающейся разработкой ПО, представлена у Гринтера [Grinter 1999].)

В частности:

- Архитектор-строитель может обратиться к тысячелетней истории и посмотреть, как работали архитекторы прошлого. Он может посещать и изучать здания, которые стоят сотни, а иногда и более тыся-

чи лет, и все еще используются. История программной отрасли насчитывает всего несколько десятков лет, а наши разработки часто не предназначены для публики. Кроме того, у архитекторов-строителей давно существуют стандарты для описания чертежей и спецификаций, благодаря чему современные архитекторы могут пользоваться документированной историей архитектуры.

- Здания являются материальными продуктами; существуют принципиальные различия между планами, созданными архитектором, и зданием, построенным рабочими.

В крупных программных проектах часто участвует много архитекторов. Некоторые архитекторы обладают узкой специализацией (скажем, базы данных или сети) и обычно работают в составе групп, но в тексте книги мы будем предполагать, что архитектор только один.

Из чего складывается программная архитектура?

Было бы неправильно относиться к архитектуре как к простой сущности, которая может быть описана одним документом или рисунком. Архитектору приходится принимать много решений. Чтобы эти решения приносили пользу, их необходимо документировать для последующего анализа, обсуждения, изменения и утверждения, после чего они используются для принятия решений и конструирования продукта. В программных системах у этих проектировочных решений имеются поведенческие и структурные аспекты.

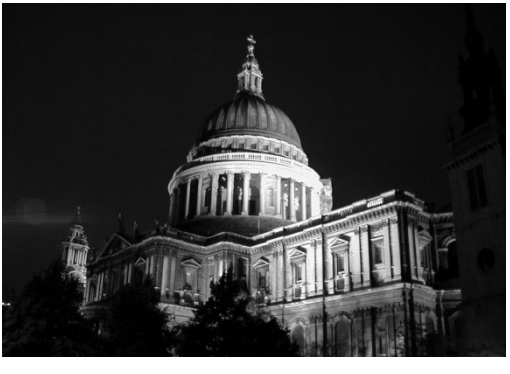
Внешние поведенческие описания показывают, как продукт будет взаимодействовать со своими пользователями, другими системами и внешними устройствами. Эти описания должны иметь форму требований. Структурные описания показывают, как продукт делится на части и как эти части связаны друг с другом. Внутренние поведенческие описания представляют интерфейсы между компонентами. В структурных описаниях часто демонстрируются несколько альтернативных представлений одной части, потому что всю информацию невозможно содержательно изложить в одном рисунке или документе. Компонент в одном представлении может быть составной частью компонента в другом представлении.

Программные архитектуры часто представляются в виде многоуровневых иерархий, склонных к смешению нескольких разных структур на одной диаграмме. В 1970-е годы Парнас указал, что термин «иерархия» стал модным словечком, которое часто применяется некорректно, затем дал точное определение термина и привел несколько примеров структур, используемых в разных целях при проектировании разных систем (Parnas 1974). Описание архитектурных структур в виде

Повторное использование архитектуры

Для поддержания огромного купола Софийского собора (верхняя иллюстрация), построенного в Константинополе (ныне Стамбул) в VI веке, были впервые применены конструкции, называемые «парусами». Собор считается одним из шедевров византийской архитектуры.

Спустя 1100 лет Кристофер Рен использовал аналогичное решение при строительстве купола собора Святого Павла (нижняя иллюстрация), архитектурной достопримечательности Лондона. Оба здания до сих пор стоят и продолжают использоваться в наши дни.



набора *представлений* (*views*), ориентированных на решение разных задач, в настоящее время считается стандартной практикой в архитектуре (Clements et al. 2003; IEEE 2000). Мы будем использовать термин «архитектура» для обозначения набора прокомментированных диаграмм и функциональных описаний, которые определяют структуры, используемые для проектирования и конструирования систем. В сообществе разработчиков ПО существует много используемых и рекомендуемых форм таких диаграмм и описаний. Некоторые примеры приведены у Хоффмана и Вайсса (Hoffman and Weiss, 2000, главы 14 и 16).

Архитектура программы или компьютерной системы представляет собой совокупность структур системы, состоящих из программных элементов, внешне видимых свойств этих элементов и отношений между ними.

«Внешне видимыми свойствами» называются ожидания других элементов в отношении данного элемента: предоставляемый сервис, характеристики быстродействия, механизм обработки ошибок, использование общих ресурсов и т. д.

Лен Басс, Пол Клементс и Рик Казман,
«Software Architecture in Practice», Second Edition

Архитектура и композиция системы

Архитектура является частью композиции системы; она выводит на первый план некоторые подробности, абстрагируясь от других. Таким образом, архитектура является подмножеством композиции. Разработчик, все внимание которого сосредоточено на реализации компонента системы, может недостаточно хорошо представлять себе схему взаимодействия всех компонентов; его интересует только композиция и разработка небольшого числа компонентов, в том числе архитектурные ограничения, которые должны соблюдаться, и правила, которые они могут использовать. Следовательно, разработчик имеет дело с другим аспектом композиции системы, нежели архитектор.

Если архитектура ориентируется на отношения между компонентами и внешне видимые свойства системных компонентов, то композиция помимо этого ориентируется на внутреннюю структуру этих компонентов. Например, если один набор компонентов состоит из модулей, скрывающих информацию, то внешне видимые свойства образуют интерфейсы этих компонентов, а к внутренней структуре относятся структуры данных и передача управления внутри модуля (Hoffman and Weiss 2000, главы 7 и 16).

Создание программной архитектуры

До настоящего момента мы рассматривали «архитектуру вообще», а также исследовали сходство и различия программных архитектур в сравнении с архитектурами из других областей. Пришло время перейти от вопроса «что?» к следующему вопросу – «как?» На чем должен сосредоточить свое внимание архитектор при создании архитектуры программной системы?

Функциональность системы не является основной заботой программного архитектора.

Да, вы не ошиблись – функциональность системы не является основной заботой программного архитектора.

Допустим, вас наняли для разработки архитектуры «веб-приложения». Начнете ли вы расспрашивать о макете страниц и деревьях навигации или же зададите следующие вопросы:

- Кто предоставит хостинг? Существуют ли в среде хостер-провайдера ограничения на применяемые технологии?
- Будет ли приложение работать в Windows Server, или в стеке LAMP?
- Сколько одновременно работающих пользователей должно поддерживать приложение?
- Насколько безопасным должно быть приложение? Существуют ли данные, которые необходимо защитить? Будет ли приложение развернуто для общего доступа в Интернете или в приватной интрасети?
- Можно ли назначить приоритеты ответам на эти вопросы? Например, можно ли считать, что количество пользователей важнее времени отклика?

В зависимости от ответов на эти и некоторые другие вопросы можно переходить к построению эскиза архитектуры системы. А ведь мы еще ни слова не сказали о функциональности приложения.

Допустим, в данном случае мы немного слукавили – область веб-приложений достаточно хорошо изучена, поэтому вы уже знали, какие решения окажут наиболее заметное влияние на архитектуру. Аналогичным образом, если бы речь зашла о телекоммуникационной системе или авиационной электронике, архитектор с опытом работы в соответствующей области уже имел бы некоторое представление о необходимой функциональности. Но, несмотря на это, факт остается фактом: вы смогли приступить к созданию системы, не беспокоясь о ее функциональности. Внимание было направлено на *качественные требования*, которые необходимо было удовлетворить.

Качественные требования определяют, каким образом должна быть предоставлена функциональность системы, чтобы она оказалась приемлемой для сторон, финансово заинтересованных в успехе системы. У этих сторон имеются определенные потребности, о которых архитектор должен позаботиться. Позднее мы обсудим проблемы, которые часто возникают, когда необходимо обеспечить наличие у системы обязательных качеств. Как говорилось ранее, архитектор среди прочего должен обеспечить соответствие системы потребностям клиента, и для упрощения понимания этих потребностей используются качественные требования.

Этот пример выделяет два важных принципа, типичных для успешных архитекторов: привлечение к работе заинтересованных сторон и ориентация как на качественные требования, так и на функциональность. Как архитектор вы должны сначала спросить у клиента, чего он хочет от системы и каковы приоритеты его потребностей. В реальном проекте также следовало бы обратиться к другим заинтересованным сторонам. Типичные заинтересованные стороны и их цели:

- Финансисты, которые желают знать, может ли проект быть завершен в пределах отведенного времени и ресурсов.
- Архитекторы, разработчики и тестеры, для которых сначала важна фаза исходного строительства, а позднее – фазы сопровождения и развития.
- Руководители проектов, которые должны организовывать группы и планировать этапы.
- Специалисты по маркетингу, использующие качественные требования для описания отличий системы от разработок конкурентов.
- Пользователи: конечные пользователи, системные администраторы и люди, занимающиеся установкой, развертыванием, техническим обеспечением и настройкой системы.
- Персонал технической поддержки, для которого важно количество и сложность обращений в их службу.

У каждой системы существуют свои качественные требования. Некоторые из них (такие как производительность, безопасность и масштабируемость) могут иметь четкое формальное определение, но другие, часто не менее важные (например, способность к изменениям, простота сопровождения и удобство использования), невозможно определить достаточно четко, чтобы от этого была хоть какая-нибудь польза. Не правда ли, странно: заинтересованные стороны желают реализовывать функции в программном коде, а не в оборудовании, чтобы при необходимости их можно было легко и быстро изменить, а потом едва упоминают возможность внесения изменений в формулировках каче-

ственных требований? От архитектурных решений зависит то, какие изменения будут вноситься легко и быстро, а какие потребуют времени и значительных затрат. Так разве не должен архитектор понимать ожидания своих клиентов в отношении таких качеств, как «способность к изменениям», так же хорошо, как он понимает функциональные требования?

Итак, архитектор разобрался в качественных требованиях заинтересованных сторон проекта. Что он должен делать после этого? Проанализировать компромиссы. Например, шифрование сообщений повышает безопасность, но ухудшает производительность. Конфигурационные файлы улучшают способность к изменениям, но могут снизить удобство пользования (если мы не сможем организовать проверку правильности конфигурации). Будем ли мы использовать стандартный формат таких файлов (например, XML), или изобретем свое собственное представление? В ходе проектирования архитектуры системы приходится принимать много непростых компромиссных решений.

Следовательно, работа архитектора должна начаться с общения с заинтересованными сторонами. В ходе этого общения архитектор выясняет качественные требования и ограничения, назначая им приоритеты. Почему бы не начать с функциональных требований? У любой системы обычно существует много возможных декомпозиций. Например, если выбрать в качестве отправной точки модель данных, вы придете к одной архитектуре; если начать с модели бизнес-процессов, архитектура может быть совершенно другой. В крайнем случае декомпозиция вообще отсутствует, а система разрабатывается как монолитный блок программного кода. Такой подход обеспечит выполнение всех функциональных требований, но, скорее всего, в этом случае не будут выполняться качественные требования – способность к изменениям, удобство сопровождения и масштабируемость.

Архитекторам часто приходится перерабатывать системы на архитектурном уровне – например, для перехода от простой схемы развертывания к распределенной, или от однопоточной модели к многопоточной (для удовлетворения требований к масштабируемости или производительности), или от жестко закодированных параметров к внешним конфигурационным файлам (потому что параметры, которые *никогда* не должны были изменяться, теперь вдруг изменились).

Обычно архитекторы умеют удовлетворять функциональные требования, но лишь немногие из них способны также позаботиться об удовлетворении качественных требований. Давайте вернемся к примеру с веб-приложением. Подумайте, сколько существует разных способов построения веб-страниц – Apache со статическими страницами, CGI, серв-

леты, JSP, JSF, PHP, Ruby on Rails, ASP.NET и т. д. Выбор одной из этих технологий как архитектурного решения значительно повлияет на возможность выполнения качественных требований. Например, выбор Ruby on Rails обеспечит быстрый выход на рынок, но может затруднить сопровождение системы, потому что как язык Ruby, так и инфраструктура Rails продолжают стремительно развиваться. А может быть, вы пишете систему веб-телефонии, и вам нужно заставить телефон «звонить». Если для выполнения требований к производительности сервер должен передавать веб-странице полноценные асинхронные события, то архитектура, основанная на сервлетах, упростит тестирование и внесение изменений.

В реальных проектах удовлетворение потребностей заинтересованных сторон требует принятия многочисленных решений, которые отнюдь не сводятся к выбору веб-технологии. Действительно ли вам нужна «архитектура» и действительно ли вам нужен «архитектор» для принятия решений? Кто должен принимать эти решения? Программист, который может принять многие решения неосознанно и косвенно, или архитектор, который принимает их абсолютно сознательно, четко представляя себе систему в целом, путь ее возможной эволюции и всех заинтересованных сторон? Как бы то ни было, у проекта появится архитектура. Будет ли она явно разрабатываться и документироваться, или же архитектура будет подразумеваться и, чтобы понять ее, придется читать программный код?

Конечно, часто выбор оказывается не столь бескомпромиссным. Но с ростом самой системы, ее сложности и количества работающих над ней людей эти ранние решения и способ их документирования начинают играть все более важную роль.

Надеемся, вы понимаете, насколько важны архитектурные решения для выполнения качественных требований, и примете эти решения сознательно, не полагаясь на то, что «архитектура сформируется сама собой по ходу дела».

А если система очень велика? Одна из причин, по которой мы применяем архитектурные принципы (скажем, принцип «разделяй и властвуй»), заключается в сокращении сложности и возможности параллельной работы. Это позволяет нам создавать все более крупные системы. Можно ли разделить саму архитектуру на части, над которыми будут параллельно работать разные люди? Джеррит Блау и Фред Брукс пишут о компьютерных архитектурах следующее:

> «...если и после применения всех приемов, направленных на то, чтобы задача стала постижимой для одного человека, архитектура все еще остается слишком масштабной и сложной, значит, планируемый продукт

> слишком сложен, и от его создания следует отказаться. Иначе говоря, компьютерная архитектура должна быть сложной настолько, чтобы ее мог понять один человек. Если один человек не способен спроектировать запланированную архитектуру, то один человек также не сможет ее понять.» (1977)

Действительно ли необходимо понимать все аспекты архитектуры для ее использования? Архитектура обеспечивает разделение ответственности, так что в большинстве случаев разработчику или тестеру, использующему архитектуру для построения или сопровождения системы, не обязательно иметь дело с архитектурой в целом – им достаточно взаимодействовать только с теми частями, которые необходимы для выполнения заданной функции. Это позволяет нам создавать системы, непостижимые для разума одного человека. Но прежде чем полностью игнорировать совет людей, построивших IBM System/360, одну из самых долговечных компьютерных архитектур, давайте разберемся, что заставило их выступить с таким утверждением.

Фред Брукс говорит, что концептуальная целостность является самым важным атрибутом архитектуры: «Лучше, если система... отражает одну совокупность идей проектирования, чем если она содержит много хороших, но независимых и нескоординированных идей»[1] (1995). Именно концептуальная целостность позволяет разработчику, уже знакомому с одной частью системы, быстро разобраться в другой части. Концептуальная целостность образуется из логической последовательности в таких областях, как критерии декомпозиции, применение шаблонов проектирования и форматы данных. Это позволяет разработчику применить опыт, полученный в ходе работы над одной частью системы, для разработки и сопровождения других частей системы. В системе действуют единые правила. При переходе от системы к «системе систем» концептуальная целостность также должна сохраняться и в архитектуре интеграции систем – например, выбором определенного архитектурного стиля (например, *канал сообщений «публикация/подписка»*) и последовательным применением этого стиля для интеграции всех систем в «систему систем».

Проблема архитектурной группы состоит в том, чтобы сохранить единую философию и единство мысли при создании архитектуры. Ограничьте группу минимальным количеством участников, организуйте высокий уровень сотрудничества с частым общением между участниками и назначьте одного-двух «благожелательных диктаторов» с правом последнего слова во всех решениях. Такая организационная схема

[1] Ф. Брукс. «Мифический человеко-месяц», Символ-Плюс, 2000.

часто встречается в успешных системах (как коммерческих, так и распространяемых с открытым кодом), а ее применение обеспечивает концептуальную целостность, которая является одним из непременных атрибутов красивых архитектур.

Хорошими архитекторами часто становятся люди, которые учатся у еще лучших архитекторов (Waldo 2006). Возможно, это объясняется тем, что некоторые концептуальные требования присущи практически всем проектам. Некоторые из них уже упоминались ранее, но ниже приводится более полный список. Каждое концептуальное требование сформулировано в виде вопроса, который архитектор должен задать себе в ходе проекта. Конечно, у некоторых систем могут существовать свои, дополнительные критические концептуальные требования.

Функциональность

Какую функциональность продукт предложит своим пользователям?

Способность к изменениям

Какие изменения могут потребоваться в программном продукте в будущем? Какие изменения маловероятны (а, следовательно, вам не нужно обеспечивать их простое внесение в будущем)?

Производительность

Какую производительность должен обеспечивать продукт?

Емкость

Сколько пользователей будет одновременно работать с системой? Какой объем данных система должна хранить для своих пользователей?

Экосистема

Как система будет взаимодействовать с другими системами в экосистеме, в которой она будет развернута?

Модульность

Как задача написания программного продукта делится на рабочие задания (модули) – и особенно модули, которые могут разрабатываться независимо друг от друга, при этом легко и точно дополняя потребности друг друга?

Удобство построения

Может ли программный продукт строиться в виде набора компонентов, реализуемых и тестируемых независимо друг от друга? Какие компоненты можно позаимствовать из других продуктов, а какие придется приобретать у внешних поставщиков?

Технологичность

Если продукт существует в нескольких версиях, то как он может разрабатываться в контексте линейки продуктов, с использованием общности версий? По какой стратегии должны разрабатываться продукты, входящие в линейку (Weiss and Lai 1999)? Какие капиталовложения потребуются на создание линейки программных продуктов? Какую предполагаемую прибыль принесет разработка разных продуктов, входящих в линейку?

В частности, возможно ли начать с разработки минимально полезного продукта, а затем разрабатывать дополнительные продукты линейки посредством добавления (и удаления) компонентов без изменения ранее написанного кода?

Безопасность

Требует ли использование продукта авторизации, должен ли продукт ограничивать доступ к своим данным, как обеспечить безопасность данных? Как защититься от атак «отказа в обслуживании» (DoS, Denial of Service) и других видов атак?

Наконец, хороший архитектор понимает, что архитектура влияет на организацию. Конуэй заметил, что структура системы отражает структуру той организации, которая ее построила (1968). Однако архитектор знает, что закон Конуэя может работать и в обратном смысле. Другими словами, хорошая архитектура может повлиять на структуру организации, чтобы повысить ее эффективность при построении систем на базе этой архитектуры.

Архитектурные структуры

Как же хороший архитектор справляется со всеми этими требованиями? Ранее мы уже упоминали о необходимости деления системы на структуры, каждая из которых определяет конкретные отношения среди определенных типов компонентов. Основная задача архитектора – структурировать систему таким образом, чтобы каждая структура помогала ответить на определяющие вопросы одного из концептуальных требований. Ключевые структурные решения разделяют продукт на компоненты и определяют отношения между этими компонентами (Bass, Clements, and Kazman 2003; Booch, Rumbaugh, and Jacobson 1999; IEEE 2000; Garlan and Perry 1995). Для любого продукта существует много структур, которые необходимо спроектировать. Каждая структура проектируется по отдельности, чтобы она могла рассматриваться как отдельное концептуальное требование. В нескольких ближайших разделах рассматриваются некоторые структуры, которые мо-

гут использоваться для требований из списка. Например, структуры сокрытия информации показывают, как система делится на рабочие задания. Они также могут использоваться в качестве плана изменений: для каждого предполагаемого изменения указывается, в какие модули эти изменения будут вноситься. Для каждой структуры описываются компоненты и отношения между ними, определяющие структуру. Для концептуальных требований из нашего списка самыми важными мы считаем именно перечисленные ниже структуры.

Структуры сокрытия информации

Компоненты и отношения: основными компонентами являются модули сокрытия информации. Каждый модуль представляет собой рабочее задание для группы разработчиков и воплощает проектировочное решение. Мы говорим, что проектировочное решение является секретом модуля, если оно может быть изменено без последствий для других модулей (Hoffman and Weiss 2000, главы 7 и 16). Основное отношение между модулями выражается формулировкой «является частью». Модуль сокрытия информации A является частью модуля сокрытия информации B, если секрет модуля A является частью секрета модуля B. Обратите внимание: секрет A должен изменяться без изменения других частей B; в противном случае A не является подмодулем согласно нашему определению. Например, во многих архитектурах имеются модули виртуальных устройств, секретом которых является взаимодействие с некоторыми физическими устройствами. Если виртуальные устройства делятся на типы, то каждый тип может образовывать подмодуль модуля виртуальных устройств. В этом случае секретом каждого типа виртуальных устройств является взаимодействие с устройствами данного типа.

Каждый модуль – это рабочее задание, включающее набор программ, которые должны быть написаны разработчиками. В зависимости от языка, платформы и среды «программой» может быть метод, процедура, функция, сценарий, макрос или другая последовательность команд, выполняемая на компьютере. Вторая структура модулей сокрытия информации основана на отношении «содержится в» между программами и модулями. Программа P содержится в модуле M, если написание P является частью рабочего задания M. Обратите внимание: каждая программа содержится в некотором модуле, потому что каждая программа должна быть частью рабочего задания некоторого разработчика.

Одни программы доступны через интерфейс модуля, другие используются в его внутренних операциях. Отношения между модулями также могут устанавливаться через интерфейсы. Интерфейс модуля состоит из совокупности допущений программ, внешних по отношению к мо-

дулю, относительно самого модуля, и совокупности допущений программ модуля относительно программ и структур данных других модулей. Говорят, что A «зависит от» интерфейса B, если изменение в интерфейсе B может потребовать изменений в A.

Структура «является частью» образует иерархию. Конечными узлами этой иерархии являются модули, не содержащие идентифицированных подмодулей. Структура «содержится в» также образует иерархию, поскольку каждая программа содержится только в одном модуле. Отношение «зависит от» не всегда образует иерархию, потому что два модуля могут зависеть друг от друга либо напрямую, либо через более длинную циклическую цепочку отношений «зависит от». Обратите внимание: отношение «зависит от» не следует путать с отношением «использует» (см. далее).

Структуры сокрытия информации образуют основу парадигмы объектно-ориентированного проектирования. Если модуль сокрытия информации реализуется в виде класса, то открытые методы класса входят в интерфейс модуля.

Концептуальные требования: структуры сокрытия информации должны проектироваться таким образом, чтобы они обеспечивали способность к изменениям, модульность и удобство построения.

Структуры «использует»

Компоненты и отношения: в соответствии с приведенным ранее определением модули сокрытия информации содержат одну или несколько программ (см. предыдущий раздел). Две программы включаются в один модуль в том и только в том случае, если они имеют общий секрет. Компонентами структуры «использует» являются программы, которые могут активизироваться независимо друг от друга. Обратите внимание: эти программы могут вызываться друг другом или оборудованием (например, функции обработки прерываний), вызов может поступить от программы из другого пространства имен (например, функции операционной системы или удаленные процедуры). Более того, вызов может происходить на любой стадии, от компиляции до времени выполнения.

Формирование структуры «использует» следует рассматривать только между программами, работающими на одной стадии привязки. Вероятно, начинать проще с программ, связываемых на стадии выполнения. Позднее также можно подумать о формировании отношений типа «использует» между программами на стадии компиляции или загрузки.

Мы говорим, что программа A использует программу B, если присутствие программы B и ее соответствие своей спецификации необходимо

для того, чтобы программа А соответствовала своей спецификации. Иначе говоря, для правильной работы программы А необходимо, чтобы программа В присутствовала и работала правильно. Отношение «использует» иногда обозначается формулировкой «требует наличия правильной версии». За дополнительными объяснениями и примером обращайтесь к главе 14 книги Хоффмана и Вайсса (2000).

Структура «использует» определяет, какие рабочие подмножества могут быть построены и протестированы в ходе создания основной программы. Желательно, чтобы свойство отношений «использует» в программных системах формировало жесткую иерархию, то есть в нем не было циклов. Если в отношении «использует» присутствует цикл, то каждая программа работает только при наличии и работоспособности всех остальных программ цикла. Построить отношение «использует» без циклов возможно не всегда, поэтому архитектор может рассматривать все программы в цикле как одну программу в контексте создания подмножеств. Подмножество либо включает всю программу, либо не включает ни одну из ее составляющих.

При отсутствии циклов в отношении «использует» в программном продукте формируется многоуровневая структура. На нижнем уровне (уровень 0) находятся все программы, не использующие другие программы. Уровень n состоит из всех программ, использующих программы уровня $n-1$ или более низких уровней. Уровни часто представляются в виде набора слоев, где каждый слой представляет один или несколько уровней отношения «использует». Группировка смежных уровней в отношениях «использует» помогает упростить представление, а также допускает случаи с небольшими циклами. Одна из рекомендаций для создания подобных группировок гласит, что программы одного уровня должны выполняться приблизительно в 10 раз быстрее и в 10 раз чаще, чем программы следующего уровня по вертикали (Courtois 1977).

Построение систем с иерархической структурой типа «использует» осуществляется по одному или нескольким слоям. Такие слои иногда называют «уровнями абстракции», но этот термин выбран неудачно. Компоненты представляют собой отдельные программы, а не целые модули, поэтому они могут ничего не абстрагировать (т. е. ничего не скрывать).

Часто крупная программная система состоит из слишком большого количества программ, чтобы в ней можно было выделить очевидные отношения «использует» между программами. В таких случаях отношение «использует» может формироваться между конгломератами программ – модулями, классами или пакетами. В подобных обобщенных описаниях теряется важная информация, но они помогают представить «общую

картину». Например, отношение «использует» можно сформировать на основе модулей сокрытия информации, но если только все программы модуля не находятся на одном уровне программной иерархии «использует», при этом будет утрачена важная информация о системе.

В некоторых проектах отношения «использует» полностью определяется только после реализации системы, потому что разработчики решают, какие программы они будут использовать, в ходе реализации. Однако архитекторы могут определить на стадии проектирования отношения «разрешены к использованию», ограничивающие выбор разработчиков. В дальнейшем мы не будем различать отношения «использует» и «разрешено к использованию».

Хорошо определенная структура «использует» формирует правильные подмножества системы и может использоваться для дальнейшего управления итеративными или поэтапными циклами разработки.

Концептуальные требования: технологичность и экосистема.

Структуры «обработка»

Структуры сокрытия информации и структуры «использует» имеют статическую природу; они существуют и во время проектирования, и во время программирования. Сейчас мы переходим к структурам времени выполнения. Компонентами, объединяемыми в структуры типа «обработка», являются процессы – последовательности событий времени выполнения, находящиеся под управлением программ (Dijkstra 1968). Каждая программа выполняется как часть одного или нескольких процессов. События в одном процессе сменяют друг друга независимо от последовательности событий другого процесса (если не считать синхронизируемые процессы, например ожидание процессом сигнала или сообщения от другого процесса). Выполнение процесса требует определенных ресурсов (скажем, памяти и процессорного времени), которые выделяются вспомогательными системами. Система может использовать фиксированное количество процессов или же создавать и уничтожать процессы во время своей работы. Обратите внимание: программные *потоки* (*threads*), реализованные в Linux, Windows и других операционных системах, подпадают под это определение процессов. Процессы являются компонентами различных отношений; некоторые примеры приводятся ниже.

Процесс дает работу

Один процесс создает работу, которая должна быть завершена другими процессами. Эта структура чрезвычайно важна для определения возможности возникновения взаимной блокировки в системе.

Концептуальные требования: производительность и емкость.

Процесс получает ресурсы

В системах с динамическим распределением ресурсов один процесс может управлять ресурсами, которые используются другим процессом, а другие процессы запрашивают и возвращают эти ресурсы. Поскольку запрашивающий процесс может запрашивать ресурсы у разных контроллеров (управляющих процессов), каждый ресурс должен иметь свой управляющий процесс.

Концептуальные требования: производительность и емкость.

Процесс совместно использует ресурсы

Два процесса могут совместно использовать общие ресурсы – принтеры, память, порты и т. д. При совместном использовании ресурсов необходимо обеспечить синхронизацию для предотвращения конфликтов. Для каждого вида ресурсов могут формироваться свои отношения.

Концептуальные требования: производительность и емкость.

Процесс содержится в модуле

Каждый процесс находится под управлением программы, а, как упоминалось ранее, каждая программа содержится в модуле. Соответственно можно считать, что каждый процесс содержится в модуле.

Концептуальные требования: способность к изменениям.

Структуры доступа к данным

Данные в системе могут делиться на сегменты таким образом, что программа, обладающая доступом к любым данным в сегменте, получает доступ ко всем данным в этом сегменте. Чтобы по возможности упростить описание, в декомпозиции должны использоваться сегменты максимального размера, для чего устанавливается дополнительное условие: если к двум сегментам обращаются одни и те же программы, то эти два сегмента объединяются. Структура доступа к данным состоит из двух разновидностей компонентов: программ и сегментов. Такое отношение обозначается формулировкой «имеет доступ» и является отношением между программами и сегментами. Считается, что максимальное ограничение доступа со стороны программ и жесткое соблюдение этих ограничений повышает безопасность системы.

Концептуальные требования: безопасность.

Сводка структур

В табл. 1.1 представлена краткая сводка упоминавшихся программных структур, способов их определения и концептуальных требований, решаемых с их помощью.

Таблица 1.1. Сводка основных структур

Структура	Компоненты	Отношения	Концептуальные требования
Сокрытие информации	Модули сокрытия информации	Является частью Содержится в	Способность к изменениям Модульность Удобство построения
Использует	Процесс	Использует	Технологичность Экосистема
Обработка	Процессы (задачи, программные потоки)	Дает работу Получает ресурсы Совместно использует ресурсы Содержится в модуле	Производительность Способность к изменениям Емкость
Доступ к данным	Программы и сегменты	Имеет доступ	Безопасность Экосистема

Хорошие архитектуры

Как говорилось ранее, архитектору постоянно приходится принимать компромиссные решения. Для заданного набора функциональных и качественных требований не существует единственно правильной архитектуры, единственного «верного решения». По своему опыту мы знаем, что перед тем как тратить деньги на построение, тестирование и развертывание системы, необходимо оценить архитектуру и определить, удовлетворяет ли она поставленным требованиям. Оценка пытается спрогнозировать выполнение концептуальных требований, упоминавшихся в предыдущих разделах (или специфических для конкретной системы).

Существует два стандартных подхода к оценке архитектуры (Clements, Kazman, and Klein 2002). В методах первой категории определяются свойства архитектуры, часто посредством моделирования или имитации одного или нескольких аспектов. Например, на базе проведенного моделирования производительности оценивается пропускная способность и масштабируемость системы, а модель дерева отказов используется для оценки надежности и доступности. В моделях других типов

метрики сложности и привязки используются для оценки способности к изменениям и удобства сопровождения.

Во второй (самой широкой) категории методов оценки архитектура оценивается по результатам опроса архитекторов. Методы структурного опроса весьма многочисленны. Например, в процессе SARB (Software Architecture Review Board), разработанном в Bell Labs, к оценке привлекаются эксперты, работающие в организации и обладающие глубокими познаниями в области телекоммуникаций и смежных областях (Maranzano et al. 2005).

В другом методе опроса – ATAM (Architecture Trade-off Analysis Method) (Clements, Kazman, and Klein 2002) – оцениваются риски того, что архитектура не удовлетворяет концептуальным требованиям. В ATAM используются сценарии, каждый из которых описывает концептуальное требование одной из заинтересованных сторон к системе. Затем архитекторы объясняют, как архитектура поддерживает каждый из сценариев.

Активный анализ – еще один тип «опросов наоборот»: архитекторы должны задать аналитикам вопросы, на которые, по их мнению, важно найти ответ (Hoffman and Weiss 2000, chap. 17). Аналитики находят ответы на эти вопросы, руководствуясь существующими архитектурными документами и описаниями. Наконец, поиск в Интернете выдает десятки опросников – от самых общих до специализированных для конкретных прикладных областей или технологических инфраструктур.

Красивые архитектуры

Все упоминавшиеся ранее методы помогают определить, можно ли считать архитектуру «достаточно хорошей» – то есть насколько вероятно, что разработчики и тестеры смогут построить на ее основе систему, удовлетворяющую функциональным и качественным требованиям заинтересованных сторон. В системах, которыми мы ежедневно пользуемся, часто встречаются примеры хороших архитектур.

Но как насчет архитектур, которые недостаточно назвать «достаточно хорошими»? Если бы существовал «Пантеон программных архитектур», то какие архитектуры украшали бы стены его галереи? Идея не такая нелепая, как может показаться, – в области линеек программных продуктов такой Пантеон существует[1]. Чтобы попасть в него, программный продукт должен удовлетворять целому ряду критериев: коммерческий успех, влияние на архитектуры других продуктов (то

[1] См. *http://www.sei.cmu.edu/productlines/plp_hof.html*.

есть архитектуру «заимствовали, копировали или похищали») и достаточный объем документации, чтобы архитектуру можно было понять, «не прибегая к слухам».

Какие критерии мы добавили бы к кандидатам в более общий «Пантеон архитектур», или «Галерею красивых архитектур»?

Прежде всего, следует понять, что в галерее выставлены не произведения искусства, а программные системы, предназначенные для использования. Поэтому начать, вероятно, следует с практичности архитектуры: она должна ежедневно использоваться множеством людей.

Но прежде чем использовать архитектуру, ее необходимо построить. Для нас представляют интерес архитектуры с четко определенной структурой типа «использует», обеспечивающие возможность поэтапного построения, чтобы на каждой итерации строительства создавалась система, пригодная к тестированию и использованию. Кроме того, искомые архитектуры должны обладать четко определенными интерфейсами модулей и внутренне присущей тестируемостью, чтобы ход их построения был предельно прозрачным и очевидным.

Также нас интересуют архитектуры, прошедшие проверку временем. Мы работаем в эру постоянных изменений технической среды. Красивая архитектура должна предвидеть необходимость в изменениях и обеспечить возможность простого и эффективного внесения ожидаемых изменений. Архитектуры, которые мы ищем, должны избежать «горизонта старения» (Klein 2005), при достижении которого затраты на сопровождение архитектуры становятся неприемлемо высокими.

И еще один признак архитектур, включаемых в число кандидатов. Они должны обладать возможностями, которые радуют разработчиков и тестеров, занимающихся построением и сопровождением системы, а также пользователей систем, построенных на основе этой архитектуры. Зачем радовать разработчиков? Потому что это упрощает их работу и повышает вероятность построения высококачественной системы. Зачем радовать тестеров? Потому что в процессе тестирования они пытаются имитировать действия пользователей. Если они будут довольны, то, скорее всего, довольны будут и пользователи. Представьте шеф-повара, недовольного своими кулинарными творениями. Скорее всего, клиенты, которым приходится есть эти творения, тоже будут недовольны.

Разные системы и предметные области предоставляют возможности для реализации специфических приятных особенностей, но концептуальная целостность – это такая особенность, которая проходит через все предметные области и радует всегда. Унифицированная архитектура проще и быстрее осваивается; изучив небольшую ее часть, вы начнете прогнозировать все остальное. Код, не отягощенный необхо-

димостью запоминать и обрабатывать особые случаи, становится более чистым и требует меньшего количества тестовых сценариев. Унифицированная архитектура не предлагает двух (или более) способов решения одной задачи, заставляя пользователя тратить время на выбор. Как сказал Людвиг Мис ван дер Роэ о хорошем дизайне, «Меньше – значит больше». Альберт Эйнштейн мог бы сказать, что красивые архитектуры просты, насколько возможно, но не более того.

С учетом этих критериев мы выдвигаем нескольких первых кандидатов для своей «Галереи красивых архитектур».

Список открывает архитектура бортового процессора A-7E[1], разработанная Научно-исследовательской лабораторией ВМС (NRL) в конце 1970-х годов и описанная у Басса, Клементса и Казмана (2003). Хотя эта конкретная система так и не пошла в производство, она удовлетворяет всем критериям. Эта архитектура оказала огромное влияние на дисциплину программной архитектуры, так как она впервые продемонстрировала отделение модулей сокрытия информации и структур типа «использует» от структур процессов в реальной системе. Она показала, что сокрытие информации может использоваться в качестве основного принципа декомпозиции сложной системы. Так как разработка архитектуры финансировалась правительством США, вся проектная документация находится в свободном доступе[2]. Архитектура содержит четко определенную структуру «использует», которая упрощает поэтапное конструирование системы. Наконец, структура модулей сокрытия информации предоставляет четкие и унифицированные критерии декомпозиции системы, что приводит к усилению концептуальной целостности. Бортовой процессор A-7E как выдающийся представитель программной архитектуры встроенных систем несомненно входит в нашу галерею.

Другой образец, который нам хотелось бы поместить в свою галерею, – программная архитектура телефонного коммутатора Lucent 5ESS (Carney et al. 1985). Модель 5ESS имела глобальный коммерческий успех и применялась для обеспечения коммутации в телефонных сетях во многих странах мира. Она установила новые стандарты производительности и доступности: одно устройство обслуживало более миллиона звонков в час, притом что время незапланированных простоев составляло менее 10 секунд в год (Alcatel-Lucent 1999). Объединяющие концепции архитектуры (например, модель «полувызовов» для управ-

[1] Модель штурмовика ВВС США – *Примеч. перев.*

[2] Например, обращайтесь к главам 6, 15 и 16 у Хоффмана и Вайсса (2000) или проведите поиск по строке «A-7E» в электронных архивах NRL (*http://torpedo.nrl.navy.mil/tu/ps*).

ления телефонными соединениями) стали стандартными шаблонами в области телефонии и сетевых протоколов (Hanmer 2001). Кроме сокращения количества обрабатываемых типов вызовов до $2n$, где n – количество протоколов вызова, шаблон «полувызова» связывает концепцию процесса, относящуюся к операционной системе, с концепцией типа вызова, относящейся к области телефонии; таким образом, формируется простое правило проектирования и вводится красивая концептуальная целостность. Группа разработчиков, состоящая из 3000 человек, развивала и совершенствовала эту систему в течение 25 лет. Архитектура 5ESS с ее успехом, долговечностью и влиянием в своей области является достойным дополнением нашей галереи.

В «Галерее красивых архитектур» найдется место и архитектуре WWW (World Wide Web), созданной Тимом Бернерсом-Ли в Европейском центре ядерных исследований (CERN) и описанной у Басса, Клементса

Кто такой архитектор?

Жарким летним днем по дороге идет странник. На своем пути он встречает человека, который дробит камни у обочины.

«Что ты делаешь?» – спрашивает он у работника.

Тот оборачивается: «Дроблю камни. Разве ты сам не видишь? Проходи и не мешай работать.»

Странник идет дальше. Вскоре он встречает второго человека, который тоже дробит камни под жарким солнцем. Человек усердно трудится, обливаясь потом.

«Что ты делаешь?» – спрашивает странник.

Человек оглядывается и улыбается.

«Зарабатываю себе на жизнь, – говорит он. – Но это нелегко. Может, у тебя найдется работа получше?»

Странник отрицательно качает головой и идет дальше. Неподалеку он встречает третьего человека с киркой. Солнце в зените, человек выбивается из сил, пот льет с него ручьями.

«Что ты делаешь?» – спрашивает странник. Человек останавливается, отпивает глоток воды, улыбается и поднимает руки к небу.

«Строю собор», – тяжело дыша, произносит он.

Странник смотрит на него и говорит: «У нас тут образовалась новая компания. Не хочешь стать в ней главным архитектором?»

и Кацмана (2003). Бесспорно, архитектура WWW достигла коммерческого успеха и безвозвратно изменила представления людей о работе в Интернете. Сама архитектура осталась практически неизменной, несмотря на создание новых приложений и появление новых возможностей. Общая простота архитектуры способствует достижению концептуальной целостности, но принятые решения (использование одной библиотеки на клиентах и серверах, создание уровневой архитектуры для разделения ответственности) обеспечили сохранение целостности архитектуры. Долговечность базовой архитектуры WWW, ее способность поддерживать новые расширения и возможности делают ее несомненно достойной для включения в нашу галерею.

Последним примером служит система UNIX. Она демонстрирует концептуальную целостность, широко используется и оказывает огромное влияние в сообществе. Концепции каналов и фильтров – удобная абстракция, позволяющая быстро создавать новые приложения.

Благодарности

Дэвид Парнас определил многие из описанных нами структур в своих работах, в том числе и в статье «Buzzword» (Parnas 1974). Джон Бентли (Jon Bentley) вдохновил нас на эту работу; он, Дебора Хилл (Deborah Hill) и Марк Клейн (Mark Klein), дали много полезных советов на основании более ранних вариантов.

Библиография

Alcatel-Lucent. 1999. «Lucent's record-breaking reliability continues to lead the industry according to latest quality report». Alcatel-Lucent Press Releases. June 2. *http://www.alcatel-lucent.com/wps/portal/NewsReleases/DetailLucent?LMSG_CABINET=Docs_and_Resource_Ctr&LMSG_CONTENT_FILE=News_Releases_LU_1999/LU_News_Article_007318.xml* (на 15 мая 2008 г.).

Bass, L., P. Clements, and R. Kazman «*Software Architecture in Practice*», Second Edition, Boston, MA: Addison-Wesley, 2003.[1]

Blaauw, G., and F. Brooks «*Computer Architecture: Concepts and Evolution*», Boston, MA: Addison-Wesley, 1997.

Booch, G., J. Rumbaugh, and I. Jacobson «*The UML Modeling Language User Guide*», Boston, MA: Addison-Wesley, 1999.

[1] Л. Басс, П. Клементс, Р. Кацман «Архитектура программного обеспечения на практике», 2-е издание, Питер, 2006.

Brooks, F. «*The Mythical Man-Month*», Boston, MA: Addison-Wesley, 1995[1].

Carney, D. L., et al. 1985. «The 5ESS switching system: Architectural overview». AT&T Technical Journal, vol. 64, no. 6, p. 1339.

Clements, P., et al. «*Documenting Software Architectures: Views and Beyond*», Boston, MA: Addison-Wesley, 2003.

Clements, P., R. Kazman, and M. Klein «*Evaluating Software Architectures*», Boston: Addison-Wesley, 2002.

Conway, M. 1968. «How do committees invent». *Datamation*, vol. 14, no. 4.

Courtois, P. J. «*Decomposability: Queuing and Computer Systems*», New York, NY: Academic Press, 1977.

Dijkstra, E. W. 1968. «Co-operating sequential processes». *Programming Languages*. Ed. F. Genuys. New York, NY: Academic Press.

Garlan, D., and D. Perry. 1995. «Introduction to the special issue on software architecture». *IEEE Transactions on Software Engineering*, vol. 21, no. 4.

Grinter, R. E. 1999. «Systems architecture: Product designing and social engineering». *Proceedings of ACM Conference on Work Activites Coordination and Collaboration* (*WACC '99*). 11–18. San Francisco, CA.

Hanmer, R. 2001. «Call processing». *Pattern Languages of Programming* (*PLoP*). Monticello, IL. *http://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/rhanmer0/PLoP2001_rhanmer0_1.pdf*.

Hoffman, D., and D. Weiss «*Software Fundamentals: Collected Papers by David L. Parnas*», Boston, MA: Addison-Wesley, 2000.

IEEE. 2000. «Recommended practice for architectural description of software intensive systems». Std 1471. Los Alamitos, CA: IEEE.

Klein, John. 2005. «How does the architect's role change as the software ages?» *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture* (*WICSA*). Washington, DC: IEEE Computer Society.

Maranzano, J., et al. 2005. «Architecture reviews: Practice and experience». *IEEE Software*, March/April 2005.

Parnas, David L. 1974. «On a buzzword: Hierarchical structure». *Proceedings of IFIP Congress*. Amsterdam, North Holland. [Reprinted as Chapter 9 in Hoffman and Weiss (2000)].

Waldo, J. 2006. «On system design». *OOPLSA'06*. October 22–26. Portland, OR.

Weiss, D., and C. T. R. Lai «*Software Product Line Engineering*», Boston, MA: Addison-Wesley, 1999.

[1] Фредерик Брукс «Мифический человеко-месяц», Символ-Плюс, 2000.

2

Повесть о двух системах: сказка для современных программистов

Пит Гудлиф

Архитектура – искусство расходовать пространство впустую.

Филипп Джонсон

Программная система напоминает город – запутанная сеть шоссе и гостиниц, объездных дорог и зданий. В городе кипит деловая жизнь; новые потоки управления постоянно рождаются, бурлят жизнью и угасают. Данные накапливаются, хранятся и уничтожаются. Город застроен разными зданиями: одни – высокие и изящные, другие – приземистые и функциональные, третьи – убогие и полуразвалившиеся. В потоке данных возникают «пробки», в нем существуют свои «часы пик» и дорожные работы. Качество программного «города» напрямую связано с тем, какой объем планирования был вложен в него.

Некоторым программным системам везет – они создаются опытными архитекторами, и их дизайн тщательно продуман. Структура таких систем создает ощущение элегантности и баланса. Благодаря четкой и логичной разметке в них легко ориентироваться. Судьба других сис-

тем складывается не так благополучно; это не города, а программные деревеньки, хаотично растущие на случайном скоплении кода. Их транспортная инфраструктура не справляется с потоками, а строения тусклы и непривлекательны. Оказавшись внутри такой системы, вы совершенно не представляете, как выбраться наружу.

Какую судьбу вы бы выбрали для своего кода? Какой программный город вам бы хотелось построить?

В этой главе я расскажу историю о двух таких программных городах. История взята из жизни, и, как и многие хорошие истории, она завершается моралью. Говорят, личный опыт – лучший учитель, но часто лучше учиться на чужом опыте. Извлекая уроки из ошибок и успехов этих проектов, вы избавите себя (и ваши программы) от множества неприятностей.

Две системы, описанные в этой главе, особенно интересны, потому что их судьба сложилась по-разному, притом что внешне они были очень похожи:

- Системы имели примерно одинаковый размер (около 500 000 строк кода).
- Обе были «встроенными» решениями в потребительских аудиоустройствах.
- Экосистема обоих проектов была достаточно зрелой и прошла много циклов выпуска продукта.
- Оба решения строились на базе Linux.
- Код был написан на C++.
- Обе системы были разработаны «опытными» программистами (которым иногда *стоило бы* действовать более ответственно).
- Архитекторами были сами программисты.

Имена персонажей этой истории были изменены, чтобы защитить ее участников (как невиновного, так и виноватого).

Беспорядочный мегаполис

Поднимайте, поднимайте, ровняйте путь,
убирайте преграду с пути народа моего.

Исаия, 57:14

Первая программная система, которую мы рассмотрим, будет называться «Беспорядочный мегаполис». Я вспоминаю о ней с нежностью

вовсе не потому, что она была хорошей или с ней было приятно работать, а потому что я извлек из нее полезный урок в части разработки программного обеспечения.

Я впервые столкнулся с Мегаполисом, когда поступил на работу в создавшую эту систему компанию. На первых порах работа казалась весьма перспективной. Я должен был присоединиться к группе, работавшей над «современной» кодовой базой на C++ для Linux; разработка этой кодовой базы продолжалась уже несколько лет. Словом, весьма заманчивая штука для каждого, кто разделяет мои специфические увлечения.

На первых порах работа шла не слишком гладко. Впрочем, вступая в группу для работы над новой кодовой базой, трудно ожидать, что все пойдет как по маслу. Однако проходили дни (и недели), но лучше не становилось. Изучение кода требовало неимоверно много времени, и в системе не было очевидных маршрутов. Это был тревожный признак. На микроуровне, при рассмотрении отдельных строк, методов и компонентов, было видно, что код запутан и плохо скомпонован. В нем не было ни единства, ни стиля, ни общих концепций, объединяющих разрозненные части. Тоже тревожный признак. Управление передавалось в системе по совершенно невообразимым и непредсказуемым путям – еще один тревожный признак. Вокруг было столько скверных «запахов кода» (Fowler 1999), что кодовая база не просто плохо пахла – она смердела, как зловонная мусорная свалка в жаркий летний день. Совершенно очевидный тревожный признак. Данные редко хранились поблизости от места их использования. В системе часто вводились какие-то хитроумные кэширующие промежуточные уровни, чтобы перетащить данные в более удобное место – снова тревожный признак.

Когда я попытался построить мысленный образ Мегаполиса, никто не смог объяснить мне его структуру; никто не знал все уровни, повороты и темные закоулки. Более того, никто толком не представлял себе, как работает система (как оказалось, это было редкое сочетание везения и героизма программистов, поддерживающих и сопровождающих эту систему). Люди знали маленькие области, над которыми они работали, но никто не имел общего представления о системе. И разумеется, никакой документации не было и в помине. Последний тревожный признак. Мне была нужна карта.

Я ввязался в печальную историю: Мегаполис оказался катастрофой городского планирования. Чтобы навести порядок, необходимо сначала разобраться в беспорядке, поэтому мы с большим трудом и самоотверженностью составили карту «архитектуры». Мы отыскали все основные магистрали, все транспортные артерии, все объездные пути,

все темные закоулки и нанесли их на одну главную диаграмму. Так мы впервые увидели, как реально выглядит программа.

Зрелище было кошмарное. Перед нами оказалось сплошное переплетение каких-то клубков и линий. Чтобы картина стала более вразумительной, мы выделили пути передачи управления цветом. Потом посмотрели еще раз.

Потрясающая, психоделическая картина. Словно пьяный паук искупался в нескольких горшках с краской, а потом сплел многоцветную сеть на листе бумаги. Диаграмма отдаленно напоминала рис. 2.1 (на рисунке представлена упрощенная версия; подробности слегка изменены, чтобы защитить виновного). Потом пришло прозрение: у нас получилась карта Лондонской подземки. На ней даже была своя кольцевая линия.

Система поставила бы в тупик даже коммивояжера. В самом деле, сходство архитектуры с Лондонской подземкой было невероятным: из любой точки системы в другую точку вело множество путей, и чаще всего было совершенно непонятно, какой из них лучший. Нередко расположенное поблизости место оказывалось недоступным; вам невольно хотелось проложить новый туннель между двумя точками. Или подняться и сесть на автобус либо идти пешком.

Такую архитектуру нельзя было назвать «хорошей» ни по каким показателям. Проблемы Мегаполиса выходили за пределы проектирова-

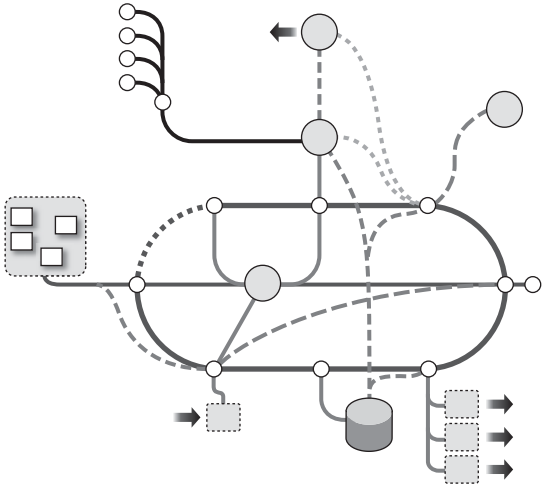


Рис. 2.1. *«Архитектура» Мегаполиса*

ния, они проникли в процесс разработки и корпоративную культуру. Они стали причиной заметного архитектурного загнивания. В течение многих лет код рос «органически» – проще говоря, никто не занимался сколько-нибудь серьезным архитектурным проектированием, а новые части попросту «лепились» на систему без особых размышлений. Никто никогда не пытался остановиться и сформировать в коде сколько-нибудь вменяемую структуру. Система росла сама по себе; перед нами был классический пример системы, проектированием архитектуры которой никто не занимался. Впрочем, нельзя было сказать, что кодовая база не имела никакой архитектуры. Архитектура была, но очень плохая.

Состояние дел Мегаполиса становилось более понятным (хотя и непростительным!) после знакомства с историей компании, создавшей систему: это была начинающая фирма, которой приходилось часто выдавать новые версии продукта. Задержки исключались – они означали финансовый крах. Программистов заставляли выдавать работоспособный код настолько быстро, насколько способен человек (и еще быстрее). Структура кода формировалась в череде бешеных «авралов».

Примечание

Плохая организационная структура компании и нездоровые процессы разработки отражаются в плохой архитектуре продукта.

Фиаско

Отсутствие планировки при строительстве Мегаполиса имело много последствий, о которых мы поговорим в этом разделе. Последствия были серьезными и выходили далеко за рамки того, что мы по наивности могли бы ожидать от плохой архитектуры. Некогда ровная колея пошла под откос.

Невразумительность

Как вы уже поняли, архитектура Мегаполиса и отсутствие спланированной структуры породили систему, разобраться в которой было очень трудно, а изменить – практически невозможно. Новые участники проекта (как, например, я) были ошарашены его сложностью и не могли понять, что же происходит.

Плохая архитектура способствовала наслоению новых плохих архитектурных решений – по сути, она не оставляла разработчикам другого выхода, потому что осмысленных путей расширения этой архитектуры не существовало. В ходе проектирования всегда выбирался путь наименьшего сопротивления для текущей задачи. Не существовало очевидных

путей исправления структурных проблем, поэтому новая функциональность просто вставлялась туда, где она создавала меньше хлопот.

Примечание

Очень важно поддерживать качественную архитектуру программного проекта. Плохая архитектура со временем разрастается.

Отсутствие связности

Компоненты системы были плохо связаны друг с другом. Вместо одной, четко определенной роли каждый компонент содержал пеструю россыпь функций, порой слабо связанных друг с другом. При взгляде на компонент было трудно понять, зачем он вообще существует, и так же трудно разобраться, где именно в системе реализуется та или иная функциональность.

Естественно, поиск и исправление ошибок в коде превратилось в сущий кошмар, что серьезно отразилось на качестве и надежности продукта.

И функциональность, и данные располагались в неверных местах системы. Многие «основные функции» системы были реализованы не в центральных блоках системы, а имитировались в периферийных модулях (ценой огромных усилий и затрат).

Дальнейшие археологические изыскания показали, почему это произошло: в исходной группе разработчиков шла активная борьба тщеславий, и некоторые ключевые программисты начали строить свои маленькие программные империи. Они захватывали функциональность, которая, по их мнению, была особенно эффектной, и втискивали в свои модули, даже если она там была неуместна. Из-за этого им приходилось создавать еще более изощренные коммуникационные механизмы, чтобы вернуть управление в нужное место.

Примечание

Здоровые отношения в группе разработки вносят непосредственный вклад в архитектуру системы. Нездоровые отношения и гипертрофированные самомнения порождают нездоровые продукты.

Избыточные привязки

В Мегаполисе не было четкого деления на слои. Зависимости между модулями не были однонаправленными, а привязки часто были двусторонними. Компонент А бесцеремонно залезал во внутренности компонента В, чтобы выполнить свою работу для одной задачи. В компоненте В присутствовали жестко закодированные вызовы функций

компонента А. В системе не было ни нижнего уровня, ни центрального «стержня». Она представляла собой монолитную массу кода.

Связность и уровень привязок

Связность и *уровень привязок* являются ключевыми свойствами программных архитектур. Не подумайте, что мы говорим о каких-то новомодных «объектно-ориентированных» концепциях; разработчики говорят о них уже много лет, еще со времен формирования идей структурного проектирования в начале 1970-х годов. При проектировании систем мы должны стремиться к использованию компонентов, обладающих следующими свойствами:

Сильная связность

Связность определяет, в какой степени в модуле объединяются родственные функции, и насколько хорошо составляющие модуля работают как единое целое. Связность – своего рода «клей», объединяющий модуль.

Слабая связность модулей является признаком некачественной декомпозиции. Каждый модуль должен иметь четко определенную роль, он не должен превращаться в «барахолку» несвязанных функций.

Низкий уровень привязок

Уровень привязок (или, для краткости, просто «привязка») является мерой взаимозависимости модулей, количества входных и выходных связей между ними. В простейших архитектурах модули обладают низким уровнем привязок, а, следовательно, в меньшей степени зависят друг от друга. Разумеется, полностью устранить привязки невозможно, иначе модули вообще не будут взаимодействовать!

Существуют разные виды взаимодействий между модулями – как прямые, так и косвенные. Модуль может вызывать функции других модулей или же его функции могут вызываться из других модулей. Модуль может использовать веб-службы или функции, публикуемые другим модулем. Он может использовать типы данных другого модуля или работать с общими данными (переменными, файлами и т. д.).

Хорошая архитектура программного продукта ограничивается абсолютным минимумом коммуникационных каналов. Они являются одной из определяющих характеристик архитектуры.

Это означало, что уровень привязок между отдельными составляющими системы был настолько высок, что «скелет» системы было невозможно построить без создания каждого отдельного компонента. Любое изменение в одном компоненте расходится кругами, вызывая изменения во многих зависимых компонентах. Компоненты кода выглядели бессмысленно в изоляции от других компонентов.

В результате низкоуровневое тестирование стало невозможным. Было невозможно не только написать модульные тесты уровня кода, но и интеграционные тесты уровня компонентов, потому что каждый компонент зависел практически от всех остальных компонентов системы. Конечно, тестирование никогда не пользовалось особенно высоким приоритетом в этой компании (у нас для него все равно не хватало времени), так что считалось, что «это не проблема». Не стоит и говорить, что продукт не отличался надежностью.

Примечание

Хорошая архитектура учитывает механизмы установления связи и количество (и природу) межкомпонентных соединений. Отдельные части системы должны быть способны к самостоятельному существованию. Жесткая привязка делает невозможным тестирование кода.

Проблемы с кодом

Недостатки высокоуровневой архитектуры проникли на уровень кода. Проблемы порождают новые проблемы (см. обсуждение разбитых окон у Ханта и Дэвиса [1999]). Из-за отсутствия единой архитектуры и «стиля» проекта никто не потрудился выработать стандарты кодирования, обеспечить использование общих библиотек или применение единых идиом. Не существовало правил назначения имен компонентов, классов и файлов. В проекте даже не существовало единой системы сборки; сценарии командного процессора и «клей», написанный на Perl, сосуществовали с make-файлами и файлами проектов Visual Studio. Компиляция этого монстра считалась «обрядом посвящения»!

Одной из самых неприметных, но серьезных проблем Мегаполиса было дублирование кода. Без четкой архитектуры и четких правил размещения функциональности программисты постоянно «изобретали велосипед» по всей кодовой базе. Простые операции, стандартные алгоритмы и структуры данных повторялись во многих модулях, причем каждая реализация обладала собственным набором неприметных ошибок и странностей. Более масштабные аспекты, такие как внешний обмен данными и кэширование, тоже были реализованы многократно.

Дополнительные археологические исследования показали, как это произошло: существование Мегаполиса начиналось с серии разнородных прототипов, которые потом объединили, вместо того чтобы выбросить. В сущности, Мегаполис был случайным образованием. Компоненты кода были принудительно сшиты, несмотря на то, что они плохо сочетались друг с другом. Со временем небрежные стежки износились, компоненты стали конфликтовать, что вызвало неизбежное трение в кодовой базе вместо гармоничной работы всех деталей.

Примечание

Нечеткая, рыхлая архитектура ведет к появлению программных компонентов, плохо написанных и плохо сочетающихся друг с другом. Кроме того, для нее характерно дублирование кода и усилий.

Внешние проблемы

Проблемы Мегаполиса вышли за пределы кодовой базы и начали сеять хаос в компании. Основные неприятности получила группа разработки, но загнивание архитектуры также отразилось на службе поддержки и на пользователях продукта.

Группа разработки

Новые участники проекта (как, например, я) были ошарашены его сложностью и не могли понять, что же происходит. Это отчасти объясняет, почему лишь немногие новички оставались в компании в течение сколько-нибудь продолжительного времени – текучесть кадров была очень высокой.

Тем, кто оставался, приходилось очень много работать, и уровень стресса был чрезвычайно высоким. Планирование новых функций наводило ужас на всю группу.

Медленный цикл разработки

Сопровождение Мегаполиса было весьма непростой задачей, поэтому даже простые изменения и исправления «незначительных» ошибок требовали непредсказуемо долгого времени. Организовать управление циклом разработки было трудно, время выполнения работы не поддавалось планированию, а цикл выпуска новых версий был медленным и неповоротливым. Клиентам приходилось подолгу ждать реализации новых важных функций, а начальство все сильнее раздражало то, что группа разработки не справляется с поставленными задачами.

Служба поддержки

Службе поддержки тоже приходилось туго – они пытались обеспечить поддержку капризного продукта и одновременно разобраться в тонкостях изменившегося поведения очередной версии.

Внешняя поддержка

Был разработан внешний протокол, который позволял другим устройствам управлять Мегаполисом в удаленном режиме. Протокол представлял собой тонкую прослойку над внутренней структурой продукта; соответственно он был вычурным, малопонятным и подверженным случайным сбоям, а пользоваться им было невозможно. Некачественная структура Мегаполиса также отравляла существование специалистам из сторонних фирм.

Внутрифирменная политика

Проблемы с разработкой привели к трениям между разными «фракциями» в компании. Группа разработки находилась в натянутых отношениях с группами маркетинга и продаж, а производственный отдел впадал в панику каждый раз, когда возникала перспектива выпуска новой версии. Руководство было в отчаянии.

Примечание

Последствия плохой архитектуры не ограничиваются кодом. Они влияют на жизнь людей, работу групп, технологические процессы и сроки исполнения.

Четкие требования

Археологические исследования выявили важную причину, из-за которой Мегаполис получился таким запутанным: в самом начале проекта *группа не знала, что она строит*.

Компания имела представление о том, какой рынок она хочет захватить, но не знала, какой продукт будет использоваться для его захвата. Решив перестраховаться, они выбрали программную платформу, которая могла бы делать много всего сразу. И притом продукт нужно было вывести на рынок как можно скорее. В спешке программисты строили безнадежно универсальную инфраструктуру, которая могла делать много всего (плохо) вместо инфраструктуры, которая хорошо выполняла бы одну задачу и могла расширяться в будущем.

Примечание

Прежде чем приступать к проектированию, важно знать, что же вы проектируете. Если вы не знаете, что должна делать ваша система, *не беритесь за нее*. Проектируйте только то, что вам действительно необходимо.

На ранних стадиях планирования Мегаполиса в проекте было слишком много архитекторов. Требования были невразумительными, поэтому каждый взял свою часть головоломки и стал работать над ней отдельно от других. Во время работы архитекторы не имели представления о проекте в целом; когда они попытались сложить собранные части, те попросту не подходили друг к другу. Времени на доработку архитектуры не было, небольшое перекрытие между компонентами программной архитектуры так и осталось – и это стало началом катастрофы городского планирования Мегаполиса.

Где он теперь?

Архитектура Мегаполиса была почти безнадежной – поверьте, мы потратили достаточно много времени в попытках исправить ее. Объем работы по переработке, рефакторингу и исправлению проблем в структуре кода был непомерно высоким. Переписывать систему? Решение не из дешевых, так как новая система обязательно должна была поддерживать старый вычурный протокол управления.

Как видите, последствием «проектирования» Мегаполиса стала дьявольская ситуация, которая неуклонно ухудшалась. Добавлять новые функции стало так сложно, что программисты только кое-как латали систему, накладывая все новые заплатки. Никто не получал удовольствия от работы с кодом, а проект заваливался «в штопор». Отсутствие архитектуры порождало плохой код, у команды пропадало рабочее настроение, а циклы разработки непрерывно удлинялись. У компании возникли серьезные финансовые проблемы.

В конечном итоге руководство признало, что Мегаполис стал экономически невыгодным, и проект был закрыт. Отважный шаг для любой организации – особенно такой, которая старается держаться на 10 шагов впереди самой себя, но при этом топчется на месте. После всего опыта C++ и Linux, накопленного группой во время работы над предыдущей версией, система была переписана на C# для Windows. Вот тебе на!

Открытка из Мегаполиса

Какой урок можно извлечь из этой истории? Плохая архитектура может иметь глубокие, серьезные последствия. К чему привело отсутствие предусмотрительности и архитектурного проектирования в Мегаполисе?

- Некачественный продукт с редким выпуском новых версий.
- Негибкая система, неспособная адаптироваться к изменениям или добавлению новой функциональности.
- Всепроникающие проблемы в коде.

- Проблемы с персоналом (стресс, отсутствие рабочего настроения, текучесть кадров и т. д.).
- Интриги и дрязги внутри компании.
- Отсутствие успеха компании.
- Сплошные неприятности и работа по вечерам.

Архитектурный городок

Форма всегда определяется функцией.

Луис Генри Салливан

Программный проект, который мы назовем Архитектурным городком, внешне был очень похож на Беспорядочный Мегаполис. Он тоже предназначался для потребительских аудиоустройств, был написан на C++ для операционной системы Linux. Однако строился он по совершенно иному принципу, а его внутренняя структура выглядела иначе.

Я участвовал в проекте Архитектурного городка с самого начала. Была сформирована новая группа способных разработчиков, которые должны были строить проект «с нуля». Группа была небольшой (изначально всего четыре программиста), и, как и в случае с Мегаполисом, в ней не было иерархии. К счастью, в группе не было межличностной напряженности, присутствовавшей в проекте Мегаполиса, и никто не стремился занять руководящее положение. Участники группы не были знакомы друг с другом и не знали, насколько хорошо они сработаются, но все мы с энтузиазмом относились к проекту и рвались в бой. Пока все неплохо.

Linux и C++ были выбраны на ранней стадии планирования, и это решение повлияло на формирование группы. С самого начала проект имел четко определенные цели: конкретный первый продукт и перспективный план будущей функциональности, которая должна быть внесена в кодовую базу. Мы должны были разработать универсальную кодовую базу, которую можно было бы применить к различным конфигурациям продукта.

В процессе разработки использовалась методология экстремального программирования (XP, eXtreme Programming) (Beck and Andres 2004). Многие считают, что она плохо сочетается с проектированием: программируй «навскидку» и поменьше задумывайся о будущем. Более того, некоторые наблюдатели пришли в ужас от нашего решения; они предсказывали, что все кончится катастрофой, как и с Мегаполисом.

Однако это распространенное заблуждение. Методология XP не препятствует проектированию; она препятствует выполнению работы, без которой можно обойтись, – это называется принципом YAGNI («You Aren't Going To Need It», Вам это не понадобится). Но там, где предварительное проектирование действительно необходимо, XP заставляет разработчиков выполнить его. Кроме того, XP поощряет построение быстрых прототипов и проверку архитектурных решений. Оба аспекта чрезвычайно важны и оказывают значительное влияние на итоговую архитектуру продукта.

Первые шаги в Архитектурный городок

На ранней стадии проектирования были определены основные области функциональности (базовый аудиоблок, управление содержанием и пользовательский интерфейс/управление). Мы проанализировали место каждой из них в системе и сформировали начальную версию архитектуры, включая многопоточные модели, необходимые для удовлетворения требований к быстродействию.

Была построена обычная иерархическая диаграмма, на которой обозначены относительные позиции разных частей системы. Одна из частей этой диаграммы в упрощенном виде изображена на рис. 2.2. Обратите внимание: мы *не* пытались заранее спроектировать всю систему до мелочей. Это была намеренно упрощенная концептуальная модель Архитектурного городка: всего несколько блоков на диаграмме, простейшая системная архитектура, которая может легко разрастаться с добавлением новой функциональности. Несмотря на свою простоту, исходная архитектура заложила прочную основу для роста. Если в Мегаполисе никто не имел представления о системе в целом, а функциональность приделывалась (или лепилась) там, где было «удобно», в на-

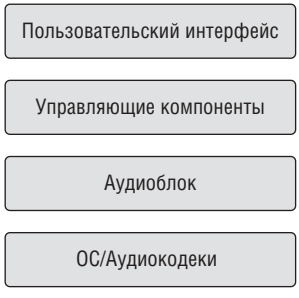


Рис. 2.2. *Исходная архитектура Архитектурного городка*

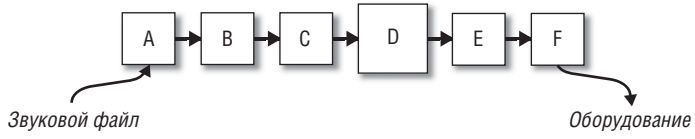


***Рис. 2.3.** Аудиоконвейер Архитектурного городка*

шей системе существовала четкая модель того, что где должно находиться.

Особое внимание в ходе проектирования было уделено «сердцу» системы: аудиоблоку. В сущности, это была внутренняя субархитектура всей системы. Чтобы определить ее, мы проанализировали потоки данных через серию компонентов и пришли к конвейерной архитектуре, изображенной на рис. 2.3. Продукты использовали разные конвейеры в зависимости от их физической конфигурации. Поначалу конвейер оставался обычной концепцией – набором блоков на диаграмме. Мы еще не решили, как это все будет связано между собой.

Также на ранней стадии были выбраны вспомогательные библиотеки, которые будут задействованы в проекте (например, библиотеки Boost C++, доступные по адресу *http://www.boost.org*, а также библиотеки баз данных). Тогда же были приняты основополагающие решения, обеспечивающие простой и связный рост кодовой базы, в том числе:

- Верхний уровень файловой структуры.
- Правила выбора имен.
- Стиль представления.
- Стандартные идиомы программирования.
- Выбор инфраструктуры модульного тестирования.
- Вспомогательная инфраструктура (контроль версий, система сборки, непрерывная интеграция).

Все эти «мелочи» были очень важны: они вплотную смыкались с программной архитектурой и, в свою очередь, повлияли на многие последующие архитектурные решения.

История продолжается

После того как группа сформировала исходную версию архитектуры, проект Архитектурного городка продолжал развиваться по правилам методологии XP. Проектирование и написание кода осуществлялось либо в парах, либо проходило тщательный анализ для проверки правильности кода.

Архитектура и код развивались и крепли. История Архитектурного городка развивалась; то внимание, которое с первых дней уделялось архитектуре, стало приносить плоды.

Определение места для новой функциональности

Благодаря четкому представлению о структуре системы с самого начала ее существования новые блоки функциональности размещались в правильных функциональных областях кодовой базы. У разработчиков никогда не возникало вопросов, где должен находиться тот или иной код. Также легко можно было найти реализацию любой существующей функции, чтобы расширить ее или исправить ошибку.

Иногда разместить код в «правильном» месте оказывается сложнее, чем просто втиснуть его в более удобное, но менее подходящее место. Существование архитектурного плана порой затрудняло труд разработчиков. Однако эти дополнительные усилия значительно упрощали жизнь позднее, в ходе сопровождения или расширения – в системе почти не было хлама, о который можно было ненароком споткнуться.

Примечание

Качественная архитектура упрощает поиск места функциональности для ее добавления/изменения или исправления ошибок. Она определяет правила размещения готового кода, а также помогает ориентироваться в системе.

Логическая целостность

Вся система была целостной. Каждое решение на каждом уровне принималось в контексте общей архитектуры. Разработчики сознательно действовали так с самого начала, чтобы весь создаваемый код полностью соответствовал архитектуре и всему остальному написанному коду.

Несмотря на то, что в ходе работы над проектом на всех уровнях кодовой базы – от отдельных строк кода до структуры системы – вносились многочисленные изменения, все они соответствовали исходным архитектурным шаблонам.

Примечание

Качественно спроектированная архитектура ведет к логически целостной системе. Все решения должны приниматься в архитектурном контексте.

Хороший вкус и элегантность нисходящего проектирования естественным образом проникали на нижние уровни. Даже на нижних уровнях код был последовательным и аккуратным. Четко определенная архитектура гарантировала, что в кодовой базе не будет дубликатов,

повсюду будут применяться знакомые шаблоны проектирования и интерфейсные идиомы, в системе не будет объектов со странным жизненным циклом или проблем с управлением ресурсами. Строки кода писались в контексте общего «городского плана».

Примечание

Четкая архитектура способствует сокращению дублирования функциональности.

Расширение архитектуры

Со временем в «общей картине» появились совершенно новые функциональные области – например, управление памятью и поддержка внешних устройств. Для проекта Мегаполис это стало бы сокрушительным ударом, а реализовать новые возможности было бы невероятно сложно. Но в Архитектурном городке дело обстояло иначе.

Системная архитектура, как и код, была пластичной и пригодной к переработке. Один из основных принципов группы разработки заключался в сохранении предельной гибкости – ничто не должно быть «выбито на камне», чтобы архитектуру можно было изменить при необходимости. Это заставляло нас выбирать простые и динамичные архитектурные решения. Соответственно код мог быстро расти, сохраняя при этом хорошую внутреннюю структуру. Включение новых функциональных блоков не создавало проблем.

Примечание

Программная архитектура не задается раз и навсегда. Будьте готовы изменить ее, если потребуется. Но чтобы архитектура могла изменяться, она должна оставаться простой. Не вносите изменения, которые нарушают простоту.

Отложенные архитектурные решения

Одним из принципов XP, действительно повысивших качество архитектуры проекта, был принцип YAGNI (не делайте того, без чего можно обойтись). В исходную архитектуру закладывались только самые важные моменты, а все остальные решения откладывались до того, как у нас появится более четкое представление о реальных требованиях и их месте в системе. Это чрезвычайно мощный подход к проектированию, который оставляет архитектору максимальную свободу выбора.

- Едва ли не худшее, что можно сделать для проекта, – проектировать то, что вы еще не понимаете. Принцип YAGNI заставляет отложить решение до того момента, когда вам станет ясно, в чем заключается проблема и как она связана с архитектурой. Это исключает из процесса элемент догадок и обеспечивает правильность архитектуры.

- Рискованно включать в программную архитектуру все, что вам может когда-либо понадобиться (включая кухонную раковину), в момент ее создания. Большая часть работы по проектированию окажется напрасной тратой сил; она лишь создаст балласт, который придется поддерживать на протяжении всего жизненного цикла изменяющегося продукта. Затраты возрастают на ранней стадии и увеличиваются на протяжении всего жизненного цикла проекта.

Примечание

Отложите архитектурные решения до того момента, когда их принятие становится абсолютно необходимым. Не принимайте архитектурные решения, пока не будете знать соответствующие требования. Не гадайте.

Поддержание качества

С самого начала в проекте Архитектурного городка были задействованы механизмы контроля качества:

- Парное программирование.
- Рецензирование кода/архитектуры для всего, что не относилось к парному программированию.
- Модульные тесты для каждого фрагмента кода.

Эти механизмы защищали систему от некорректных, неуместных изменений. Все, что не соответствовало программной архитектуре, отвергалось. На первый взгляд такой подход может показаться излишне жестким, но этого требовали процессы, на которые согласились все разработчики.

Их согласие выделяет важное обстоятельство: разработчики верили в архитектуру и считали ее достаточно важной для того, чтобы ее защитить. Они приняли на себя личную ответственность за архитектуру.

Примечание

Поддерживайте качество архитектуры. Это возможно только в том случае, если разработчики получают и принимают ответственность за нее.

Управление «техническими долгами»

Невзирая на все меры контроля качества, разработка Архитектурного городка велась весьма прагматично. С приближением срока сдачи нам иногда приходилось «срезать углы», чтобы продукт был готов в положенное время. В кодовую базу вводились небольшие кодовые «грешки», или «архитектурные бородавки», – иногда для того, чтобы функ-

циональность быстрее заработала, иногда для предотвращения рискованных изменений перед выпуском.

Но в отличие от проекта Мегаполис, эти огрехи помечались как *технические долги*, а их исправление вносилось в план. Они «торчали» на общем фоне, и разработчики не чувствовали себя спокойно до тех пор, пока не разбирались с ними. Мы снова видим, как разработчики берут на себя ответственность за качество архитектуры.

Модульные тесты формируют архитектуру

Одно из важнейших решений по поводу кодовой базы (также предписанных методологией XP) состояло в том, что весь код должен проходить модульное тестирование. Модульное тестирование обладает многими преимуществами; в частности, оно позволяет изменять часть кода, не беспокоясь о том, что это нарушит работу всех остальных частей. Некоторые области внутренней структуры Архитектурного городка подвергались весьма радикальной переработке, и модульные тесты давали нам уверенность в том, что в остальных областях системы ничего не «сломается». Например, принципиальные изменения были внесены в потоковую модель и интерфейс подключения к аудиоконвейеру. Это были серьезные архитектурные изменения, вносимые на относительно поздней стадии разработки подсистемы, но остальной код, взаимодействующий с аудиоблоком, продолжал работать идеально. Модульные тесты дали нам возможность изменять архитектуру.

Со временем Архитектурный городок становился все более зрелым проектом, и подобные «крупные» изменения происходили все реже. После переработки архитектуры ситуация стабилизировалась, и в дальнейшем происходили лишь небольшие архитектурные изменения. Система развивалась быстро, по итеративному принципу. В ходе каждого этапа архитектура совершенствовалась, пока не пришла к относительно стабильному состоянию.

Примечание

Хороший набор автоматизированных тестов для вашей системы позволяет вносить фундаментальные архитектурные изменения с минимальным риском. Он создает пространство для вашей работы.

Другое крупное преимущество модульных тестов – их несомненное влияние на архитектуру: тесты практически заставляют разработчика формировать хорошую структуру кода. Каждый мелкий компонент кода строится как четко определенная сущность, способная к независимому существованию, потому что компонент должен строиться в контексте модульного теста без построения остальных частей системы. На-

писание модульных тестов гарантирует внутреннюю связность и низкий уровень привязки компонентов с другими компонентами системы. Модульные тесты заставляют нас тщательно продумывать интерфейс каждого модуля, а также следить за его содержательностью и внутренней непротиворечивостью.

Примечание

Модульное тестирование улучшает качество архитектуры, поэтому удобство тестирования следует учитывать при проектировании.

Сроки проектирования

Одним из факторов, обусловивших успех Архитектурного городка, стали разумные сроки разработки. Они не были ни слишком короткими, ни слишком длинными. Для процветания проекта необходима благоприятная среда.

Если выделить слишком много времени, программисты часто пытаются создать Великий Шедевр (который обычно всегда *почти* готов, но почему-то так и не воплощается в реальность). Небольшое давление полезно, а ощущение срочности помогает довести дело до конца. Но если времени будет слишком мало, то создать сколько-нибудь приличную архитектуру не удастся, и вы получите недоделанное, сделанное наспех решение – как это произошло с Мегаполисом.

Примечание

Хорошее планирование проекта повышает качество архитектуры. Для создания архитектурного шедевра необходимо выделить достаточно времени – шедевры не рождаются за минуту.

Работа с архитектурой

Кодовая база проекта была большой, но осмысленной и логичной. Новые программисты легко разбирались в ней и включались в работу. В системе не было ни лишних сложных связей, ни противного унаследованного кода, с присутствием которого приходится считаться.

Код не создавал особых проблем, с ним было приятно работать, поэтому текучесть кадров в группе была очень, очень низкой. Отчасти это было связано с тем, что разработчики взяли на себя ответственность за архитектуру и постоянно желали улучшать ее.

Интересно проследить за тем, как развитие группы разработки следовало за архитектурой. Принципы проекта «Архитектурный городок» требовали, чтобы у областей архитектуры не было «владельцев», то есть

любой разработчик мог работать с любой частью системы. От каждого участника ожидали качественного кода. Если мешанина Мегаполиса создавалась трудами множества нескоординированных, мешающих друг другу программистов, то Архитектурный городок был чистым и логичным, а его тесно взаимодействующие компоненты создавались тесным взаимодействием коллег. Во многих отношениях закон Конуэя[1] работал в обратную сторону: группа формировалась в процессе формирования продукта.

Примечание

Организация группы оказывает неизбежное влияние на создаваемый код. Со временем архитектура также начинает влиять на совместную работу группы. Если участники разобщены, взаимодействие кода оставляет желать лучшего. Если они работают дружно, то архитектура хорошо интегрируется.

Где он теперь?

Через некоторое время архитектура проекта выглядела так, как показано на рис. 2.4. Иначе говоря, она сохранила большое сходство с исходной архитектурой, с несколькими заметными изменениями – и со значительным практическим опытом, доказывавшим правильность архитектуры. Здоровый процесс разработки; небольшая группа разработчиков, серьезно относящихся к результатам своего труда; должное внимание к логической целостности – все это привело к невероятно

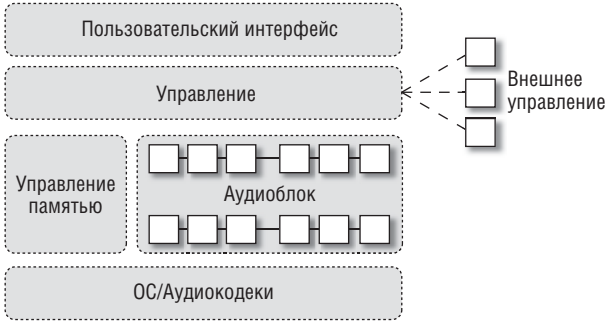


***Рис. 2.4.** Итоговая архитектура проекта*

[1] Закон Конуэя утверждает, что структура кода соответствует структуре группы. В упрощенной формулировке он гласит: «Если над компилятором работают четыре группы, то вы получите четырехпроходной компилятор».

простой, ясной и последовательной архитектуре. Простота сработала во благо Архитектурного городка, обернувшись пластичностью кода и ускорением цикла разработки.

На момент написания книги проект «Архитектурный городок» существовал уже более трех лет. Кодовая база реально использовалась и породила ряд успешных продуктов. В настоящее время она все еще разрабатывается, все еще растет и расширяется, ежедневно изменяется. В следующий месяц ее архитектура может радикально измениться по сравнению с текущим месяцем – но скорее всего, этого не произойдет.

Хочу сразу заявить, что код ни в коем случае не идеален. В нем присутствуют «технические долги», нуждающиеся в доработке, но они бросаются в глаза на фоне общей аккуратности и будут непременно исправлены в будущем. Ничто не задается раз и навсегда, а благодаря адаптивной архитектуре и гибкой структуре кода все недостатки могут быть устранены. Почти все компоненты находятся там, где им положено находиться, потому что архитектура хорошо продумана.

Что дальше?

...Когда же настанет совершенное, тогда то, что отчасти, прекратится.

К коринфянам, 13:10

Конечно, простая история о двух программных системах не является исчерпывающе полным трактатом о программных архитектурах. Но я показал, что архитектура оказывает глубокое влияние на судьбу программного проекта. Архитектура влияет практически на все, что имеет к ней хотя бы отдаленное отношение, определяя состояние кодовой базы и смежных областей. Подобно тому как преуспевающий город приносит процветание и известность своему округу, хорошая программная архитектура может принести успех своему проекту и всем, кто от нее зависит.

Хорошая архитектура обусловлена многими факторами. Некоторые из них перечислены ниже (список не полон):

- Уделяйте время и силы сознательному предварительному проектированию. (Многие проекты терпят неудачу в этом отношении еще до начала работы.)
- Важны квалификация и опыт разработчиков. (Прошлые ошибки помогают выбрать правильное направление в следующий раз! Несомненно, проект Мегаполис меня кое-чему научил.)

- Постоянно удерживайте архитектуру в поле зрения во время работы над проектом.
- Группа получает и принимает ответственность за общую архитектуру продукта.
- Не бойтесь изменять архитектуру: ничто не задается раз и навсегда.
- Группа должна иметь правильно подобранный состав (проектировщики, программисты, руководители) и численность. Проследите за тем, чтобы в группе сформировались здоровые рабочие отношения, поскольку эти отношения оказывают неизбежное влияние на структуру кода.
- Архитектурные решения должны приниматься только при наличии всей необходимой информации. Отложите на будущее те решения, которые не могут быть осознанно приняты сейчас.
- Необходимы грамотное управление проектом и правильно выбранные сроки.

Ваш ход

Не теряйте священной любознательности.

Альберт Эйнштейн

Вы читаете эту книгу, потому что вам небезразлична программная архитектура и вы хотели бы улучшить свои программы. Вам предоставляется отличная возможность. Ответьте на несколько простых вопросов о тех системах, с которыми вы имели дело в прошлом.

1. Какой была лучшая из известных вам системных архитектур?
   - Почему вы решили, что это хорошая архитектура?
   - Какие последствия имела эта архитектура – как внутри кодовой базы, так и за ее пределами?
   - Чему вы научились на ее примере?
2. Какой была худшая из известных вам системных архитектур?
   - Почему вы решили, что это плохая архитектура?
   - Какие последствия имела эта архитектура – как внутри кодовой базы, так и за ее пределами?
   - Чему вы научились на ее примере?

Библиография

Beck, Kent, with Cynthia Andres «*Extreme Programming Eplained*», Second Edition. Boston, MA: Addison-Wesley Professional, 2004.[1]

Fowler, Martin «*Refactoring: Improving the Design of Existing Code*». Boston, MA: Addison-Wesley Professional, 1999.[2]

Hunt, Andrew, and David Thomas «*The Pragmatic Programmer*». Boston, MA: Addison-Wesley Professional, 1999.[3]

[1] Кент Бек «Экстремальное программирование». – Пер. с англ. – СПб: Питер, 2004.

[2] Мартин Фаулер «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

[3] Э. Хант, Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». – Пер. с англ. – Лори, 2009.

II

Архитектура корпоративных приложений

	Принципы и свойства		Структуры
✓	Гибкость		Модуль
✓	Концептуальная целостность	✓	Зависимость
	Возможность независимого изменения		Обработка
	Автоматическое распространение	✓	Доступ к данным
	Удобство построения		
✓	Адаптация к росту		
	Сопротивление энтропии		

3

Масштабирование

Джим Уолдо

Введение

Одной из самых интересных проблем при проектировании системной архитектуры является сохранение гибкости в выборе масштаба системы. В наши дни роль масштабирования непрерывно растет, так как все больше систем работает в интрасетях или в Интернете. В таких системах идея планирования мощности выглядит абсурдно, если вы хотите, чтобы допустимая погрешность составляла меньше двух порядков. Созданный вами сайт неожиданно становится популярным, и счет посетителей буквально идет на миллионы. Так же просто (и с такими же катастрофическими последствиями) возможен и другой вариант: вы создаете сайт, которым никто особенно не интересуется. Дорогостоящее оборудование простаивает, поглощая деньги за потребляемую энергию и администрирование. В сетевом мире переход сайта из одного состояния в другое происходит за минуты.

С проблемами масштабирования сталкивается каждый, кто подключает свою систему к сети. Эти проблемы особенно актуальны для массовых многопользовательских онлайновых игр (MMO) и виртуальных миров. Такие системы должны масштабироваться в расчете на боль-

шое количество пользователей. Однако, в отличие от веб-серверов, на которых пользователи запрашивают более или менее статическую информацию и не взаимодействуют друг с другом, игроки ММО или обитатели виртуальных миров приходят ради взаимодействий как с миром (изменение базовой информации о мире), так и друг с другом. Взаимодействия усложняют масштабирование инфраструктур таких систем, поскольку взаимодействие пользователей с системой в основном происходит независимо (хотя, конечно, бывают и исключения) и не приводит к значительному изменению состояния виртуального мира. Если взять двух конкретных участников мира, вероятность их взаимодействия в любой отдельный момент времени ничтожно мала. Но практически каждый игрок почти всегда с кем-то взаимодействует. Таким образом, система получается массово-параллельной, но взаимозависимой в малом количестве взаимодействий.

Масштабирование ММО и виртуальных игр дополнительно усложняется культурой, сложившейся вокруг этих систем. И ММО, и виртуальные миры происходят от видеоигр. При программировании игр для PC и игровых приставок программист традиционно мог считать, что игра работает на автономном компьютере или приставке. В такой среде все ресурсы находятся в распоряжении игровой программы, а все проблемы ограничиваются конкретным пользователем (причем ошибки в коде и странности поведения часто воспринимаются как часть логики самой игры).

Такие игры, а также компании, которые их создают и совершенствуют, являются частью индустрии развлечений. Группой разработчиков руководит продюсер, у игры существует свой сценарий и предыстория. Игра должна быть захватывающей, убедительной, но самое главное – интригующей. Надежность – качество хорошее, но едва ли обязательное. Расширяемость, то есть возможность добавления новых сюжетных линий и локаций в обновлениях игры, является свойством самой игры, а не свойством программного кода (то есть его использования новыми способами).

С расцветом онлайновых игр и виртуальных миров эта культура начала переходить в среду с требованиями, характерными для тех, с которыми сталкивается разработчик крупных коммерческих приложений. Когда несколько игроков взаимодействует с сервером по сети, сбой сервера вследствие непредвиденных действий игрока повлияет на многих других игроков. В мирах могут формироваться свои экономики (порой взаимодействующие с экономикой реального мира), поэтому стабильность и целостность виртуального мира перестает быть просто игрой. А когда счет игроков или обитателей виртуального мира идет на мил-

лионы, возможность масштабирования становится важнейшим требованием любой архитектуры.

Проект Darkstar (далее в этой главе – просто Darkstar) стал ответом на изменяющиеся потребности создателей игр и виртуальных миров. В ходе этого проекта, предпринятого исследовательской группой из Sun Microsystems Laboratories, изучалась архитектура масштабирования. Проект особенно интересен тем, что он ориентирован на создателей MMO и виртуальных миров – категорию программистов, потребности которых сильно отличаются от того, к чему привыкли мы, проектировщики систем. Итоговая архитектура выглядит знакомо, пока не начнешь ее изучать более подробно; в этот момент становится видно, что она выглядит совершенно не так, как подсказывает ваш прошлый опыт. Она обладает своеобразной красотой и наглядно демонстрирует, как при изменении требований меняется подход к построению системы.

Контекст

Архитектура системы, как и физическая архитектура здания или города, должна приспосабливаться к контексту существования артефакта, созданного на основе этой архитектуры. В физической архитектуре контекст включает в себя историческое окружение, климат, возможности местных художников и доступные строительные материалы, а также предполагаемое использование здания. В программной архитектуре контекст включает не только приложения, которые будут использовать архитектуру, но и программистов, работающих с архитектурой, и ограничения, действующие в созданных системах.

При построении архитектуры Darkstar мы[1] прежде всего поняли, что любая архитектура, ориентированная на масштабирование, должна быть развернута на множестве компьютеров. Даже самые крупные современные компьютеры вряд ли справятся с потребностями современных онлайновых игр (например, *World of Warcraft* имеет пять миллионов подписчиков, сотни тысяч из которых могут быть активны в отдельный момент времени). Даже если бы существовал компьютер, способный справиться с такой нагрузкой, было бы экономически нере-

[1] Говоря о разработке архитектуры проекта Darkstar, я в основном употребляю множественное число – «мы» вместо «я». Это нечто большее, чем простой литературный прием. Разработка архитектуры стала плодом совместных усилий. Проект начинали Джеффри Кессельман (Jeffery Kesselman), Сет Проктор (Seth Proctor) и Джеймс Мегуайр (James Megquier), а в текущее состояние его привели Сет, Джеймс, Тим Блэкман (Tim Blackman), Энн Уолрат (Ann Wollrath), Джейн Лиозо (Jane Liozeaux) и я.

ально ожидать, что игра окажется настолько успешной, чтобы оправдать такие изначальные затраты на оборудование. Подобные приложения должны начинать с малого, постепенно наращивать мощность с ростом пользовательской базы и снижать ее по мере падения интереса к игре. Такая схема хорошо соответствует концепции распределенной системы, в которой компьютеры (относительно небольшие) добавляются по мере роста спроса и отключаются с его снижением. Итак, мы с самого начала знали, что архитектура в целом должна представлять собой распределенную систему.

Мы также знали, что система должна использовать современные тенденции микросхемной архитектуры. MMO и (в меньшей степени) виртуальные миры традиционно использовали в отношении масштабирования закон Мура. С удвоением скорости процессора удваивается сложность, богатство и интерактивность создаваемого мира. Ни в одной другой области обработки данных преимущества роста скорости процессоров не использовались так интенсивно, как в игровом мире. Для игровых персональных компьютеров всегда были характерны предельно возможная скорость процессора, объем памяти и графические возможности. Игровые приставки еще агрессивнее штурмуют эти ограничения: их графические системы значительно превосходят возможности даже мощных рабочих станций, а вся машина строится на базе специфических потребностей игрока.

Последние достижения в эволюции процессоров, от постоянного роста тактовой частоты до конструирования многоядерных процессоров, изменили динамику игровых возможностей. Вместо того чтобы ускорять выполнение одной операции, новые процессоры создаются для одновременного выполнения нескольких операций. Введение параллелизма на аппаратном уровне повышает общую производительность, если выполняемые чипом операции действительно могут выполняться одновременно. При неизменной тактовой частоте процессор с четырьмя ядрами может выполнять в четыре раза больше работы, чем одноядерный процессор. На практике ускорение носит нелинейный характер, поскольку в других частях системы параллелизм не приводит к подобному эффекту. Однако параллелизм способен повысить общее быстродействие системы, а процессоры для подобного параллельного выполнения строятся гораздо проще, чем процессоры с повышением тактовой частоты.

На первый взгляд MMO и виртуальные миры являются хорошими кандидатами для многоядерных процессоров и распределенных систем. Большая часть того, что происходит в MMO или виртуальном мире (как и большая часть происходящего в реальном мире), не зависит от других событий в этом мире. Игроки выполняют свои миссии или за-

нимаются украшением своих комнат. Они сражаются с монстрами или выбирают себе виртуальные наряды. Даже во время общения с другим игроком или представителем населения мира они взаимодействуют с очень небольшим подмножеством обитателей мира. Это свойство характерно для массово-параллельных задач, а именно с такими задачами многоядерные процессоры и распределенные системы справляются особенно хорошо.

Хотя задачи в таких системах могут быть массово-параллельными, программисты, работающие над этими системами, не искушены в тонкостях распределенного или параллельного программирования. Эти области чрезвычайно нетривиальны; они сложны даже для людей с хорошей подготовкой и опытом работы. Требовать от игровых программистов разработки распределенного игрового сервера с высокой степенью параллелизма значило бы требовать от них того, что выходит за рамки их опыта и квалификации.

Первая цель

Контекст определил первую цель нашей архитектуры. Требования к масштабированию означали, что система должна быть распределенной и параллельной, но разработчику игры нужно было предоставить значительно упрощенную модель программирования. Проще говоря, разработчик должен воспринимать систему как отдельный компьютер, на котором работает один программный поток, а все механизмы, обеспечивающие работу многих потоков на множестве компьютеров, должны быть скрыты инфраструктурой проекта Darkstar.

В общем случае скрыть распределенность или параллелизм от приложения невозможно. Но MMO и виртуальные миры не относились к общему случаю. За изоляцию, к которой мы стремились, приходилось расплачиваться обязательным применением крайне конкретной и ограниченной модели программирования. К счастью, именно такая модель хорошо соответствует типу программирования, уже применяемому в серверных компонентах игр и виртуальных миров.

Общая модель программирования Darkstar должна была быть *реактивной*: серверная сторона игры пишется в виде получателя событий, генерируемых клиентами (то есть компьютерами, используемыми игроками, – PC или игровыми приставками). Обнаружив событие, игровой сервер генерирует задачу – кратковременную последовательность действий, которая включает обработку данных в виртуальном мире и взаимодействие с клиентом, сгенерировавшим исходное событие (и, возможно, с другими клиентами). Задачи также могут генерироваться самим игровым сервером – либо в ответ на некоторое внутрен-

нее изменение, либо на периодической, регулярной основе. При такой схеме игровой сервер может генерировать персонажей игры или мира, не находящихся под управлением внешнего игрока.

Такая программная модель хорошо соответствует специфике игр и виртуальных миров, но она также применяется во многих крупных коммерческих архитектурах, например в J2EE и в веб-службах. Потребность в построении архитектуры, отличной от этих механизмов, была обусловлена спецификой среды, в которой существуют ММО и виртуальные миры. Эта среда является практически полной противоположностью среды применения этих архитектур; таким образом, все, что вам известно о корпоративных средах, в новом мире окажется неверным.

Согласно традиционным представлениям в корпоративной среде тонкий клиент обычно подключается к толстому серверу (который, в свою очередь, часто подключается к еще более толстому серверу баз данных). Сервер хранит большую часть информации, необходимой клиентам, и играет роль фильтра при обращениях к служебной базе данных. На стороне клиента хранится минимальный объем информации состояния; в лучшем случае клиентский компьютер обладает минимумом памяти, не имеет собственного диска и представляет собой «интеллектуальное устройство вывода» для работы с сервером, на котором выполняется большая часть реальной работы.

Игровой мир

Среда ММО и виртуальных миров начинается с очень толстого клиента. Как правило, это мощный PC с современным процессором, большим объемом памяти и видеокартой, которая сама по себе обладает немалой вычислительной мощью, или игровая приставка, специально спроектированная для интенсивной работы с графикой и интерактивных задач. Как можно больший объем данных вытесняется на сторону клиентов, особенно если эти данные неизменны (географические сведения, текстуры, правила). Сервер остается по возможности простым; в общем случае на нем хранится крайне абстрактное представление мира и находящихся в нем сущностей. Кроме того, в соответствии с архитектурой сервер должен выполнять как можно меньше вычислений. Большая часть обработки данных происходит на стороне клиента. Настоящая функция сервера заключается в хранении общих эталонных сведений о состоянии мира; наличие эталона гарантирует, что любые отклонения в состоянии мира на стороне клиента будут исправляться. Эталон должен храниться на сервере, так как люди, управляющие клиентами, являются заинтересованной стороной. Поддавшись искушению, они могли бы изменить состояние мира (если у них получится, конечно) в свою пользу. Проще говоря, игроки будут жуль-

ничать, если у них будет такая возможность, поэтому сервер должен стать источником непререкаемой истины.

Закономерности доступа к данным в MMO и виртуальных мирах тоже основательно отличаются от тех, которые мы знаем по корпоративным средам. Как правило, в корпоративных средах около 90% обращений производится только для чтения, а большинство задач требует большого объема данных для внесения небольших изменений. В MMP и виртуальных мирах большинство задач работает с минимальным объемом данных состояния на сервере, но зато примерно в половину данных, к которым они обращаются, вносятся изменения.

Задержка – главный враг

Но самые серьезные различия между двумя средами относятся к действиям пользователей. В корпоративной среде конечной целью является деловая деятельность, и небольшие задержки в обработке допустимы, если они способствуют улучшению общей производительности. В среду MMO и виртуальных миров игроки приходят, чтобы развлекаться, а задержка – враг развлечения. Следовательно, инфраструктура MMO и виртуальных миров должна строиться на требованиях минимальной задержки, даже за счет снижения общей пропускной способности системы.

Разработчики онлайновых игр и виртуальных миров уже нашли пути масштабирования для большого количества пользователей. Применяемые в настоящее время механизмы делятся на две группы. Первая группа имеет географическую природу. Игра состоит из множества областей, при этом каждая область обслуживается одним сервером. Она может соответствовать острову или комнате в виртуальном мире, городу или долине в онлайновой игре. Проектировщики игры стремятся сделать географические области по возможности независимыми друг от друга, при этом масштаб областей выбирается так, чтобы сервер не был перегружен слишком большим количеством пользователей, занимающих эту область. На практике ограничения на заполнение областей часто устанавливаются автоматически: когда сервер перегружен, игра медленнее реагирует на действия пользователя и становится менее интересной. В результате игроки уходят в другие, более интересные области, ранее перегруженная область освобождается и время отклика уменьшается.

Недостаток масштабирования на базе распределения географических областей по разным серверам заключается в том, что решение о том, какие области масштабируются на те или иные сервера, должно быть принято в момент написания игры. Добавление новых областей в игру

или виртуальный мир происходит относительно легко, но для смены области, выделенной серверу, придется изменять код продукта. Единицы масштабирования тоже должны выбираться в процессе разработки.

Второй способ борьбы с перенаселенностью областей в игре или виртуальном мире основан на применении *шардов* – копий области, обслуживаемых собственным сервером и не зависящих от других шардов, представляющих ту же часть игры. Таким образом, шард представляет дополнительную копию некоторой комнаты или деревни, позволяя вдвое большему количеству игроков занимать эту часть мира. Недостаток такого решения заключается в том, что игроки из разных шардов не могут взаимодействовать друг с другом. Для игр и миров, ориентированных на социальные взаимодействия (в отличие от простого игрового процесса), этот недостаток может стать очень существенным. Игроки стремятся не только находиться в виртуальном мире, но и общаться со своими друзьями (реальными или виртуальными). Деление на шарды противоречит этой цели.

Таким образом, другой важнейшей целью архитектуры Darkstar должна была стать возможность оперативного масштабирования, в котором не была бы задействована игровая логика. Инфраструктура должна была позволить игре динамически реагировать на нагрузку (вместо того чтобы закладывать такую реакцию в архитектуру самой игры).

Архитектура

Архитектура Darkstar строится в виде набора отдельных служб в адресном пространстве серверной части игры или виртуального мира. Каждая служба определяется небольшим программным интерфейсом. Хотя это и не входило в исходные намерения проектировщиков, основные сервисные функции Darkstar имеют много общего с сервисными функциями классической операционной системы; это позволяет серверной стороне игры или виртуального мира работать с долгосрочным хранением данных, планировать и запускать задачи, а также обмениваться данными с клиентской стороной игры или виртуального мира.

Структурирование системы в виде взаимосвязанного набора служб – очевидное начало процесса «разделяй и властвуй», играющего важную роль в архитектуре любой крупной компьютерной системы. Каждая служба характеризуется интерфейсом, который защищает пользователей службы от изменений реализации и позволяет изменять реализации независимо друг от друга. Изменения в реализации одной службы не должны влиять на реализацию другой, даже если последняя использует изменяемую реализацию (предполагается, что интерфейс и его семантика остаются неизменными).

У нас были и другие причины для выбора декомпозиции на службы. С самого начала Darkstar был задуман как проект с открытым исходным кодом в надежде, что для усиления основной группы мы сможем привлечь других участников к созданию дополнительных служб, расширяющих функциональность ядра системы. Управление сообществом открытого кода – задача непростая при любых обстоятельствах, и мы полагали, что максимальный уровень изоляции служб, образующих инфраструктуру, позволит улучшить изоляцию между разными уровнями реализации. Также на тот момент было неясно, существует ли единый набор служб, подходящих для всех MMO и виртуальных миров.

Структурирование инфраструктуры в виде набора независимых служб позволяло использовать разные наборы этих служб в разных обстоятельствах, обусловленных потребностями конкретного проекта. Службы, включаемые в конкретный стек Darkstar, могут определяться в конфигурационном файле.

Макроструктура

На рис. 3.1 изображена основная структура игры или виртуального мира на базе инфраструктуры Darkstar. Несколько серверов формируют служебную подсистему игры или виртуального мира. На каждом сервере выполняется копия выбранного набора служб (стек Darkstar) и копия игровой логики. Клиент подключается к одному из серверов для взаимодействия с абстрактным представлением игрового мира, хранящимся на сервере.

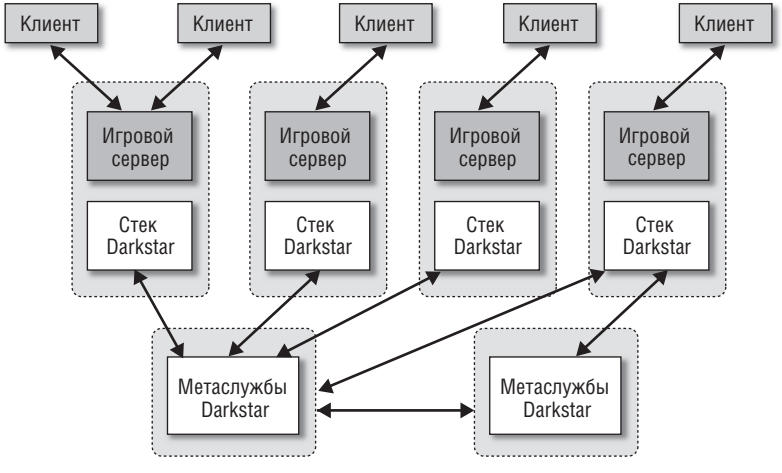


Рис. 3.1. *Высокоуровневая архитектура проекта Darkstar*

В отличие от многих репликационных схем, разные копии игровой логики не предназначены для обработки одних и тех же событий. Каждая копия способна независимо взаимодействовать с клиентами. Репликация в этой архитектуре используется в основном для того, чтобы сделать возможным масштабирование, а не для обеспечения отказоустойчивости (хотя, как мы увидим позднее, отказоустойчивость тоже обеспечивается). Кроме того, игровой логике не обязательно знать о существовании других копий сервера, работающих на других компьютерах.

Код, написанный игровым программистом, работает так, словно он выполняется на одном компьютере; координацией его копий занимается инфраструктура Darkstar. В сущности, игру на базе Darkstar можно запустить и на одном сервере, если он способен предоставить все ресурсы, необходимые для игры.

Для подключения к игровой логике клиенты используют механизмы, являющиеся частью инфраструктуры. Эти механизмы поддерживают как прямой обмен данными между клиентом и сервером, так и создание канала публикации-подписки, когда любое передаваемое по каналу сообщение доставляется только клиентам, подписавшимся на этот канал.

Координацией стеков Darkstar занимаются метаслужбы, доступные по сети и скрытые от программиста игры или виртуального мира. Эти метаслужбы позволяют разным копиям стека координировать общее состояние игры. Например, они следят за работоспособностью всех копий и инициируют процедуру восстановления в случае отказа какой-либо копии; следят за нагрузкой на копии и перераспределяют ее в случае необходимости; позволяют в любой момент подключить новые серверы для повышения мощности всей системы.

Так как службы полностью скрыты от пользователей Darkstar, их можно в любой момент изменять, добавлять и удалять без изменения кода игры или виртуального мира.

Для программиста, строящего игру или виртуальный мир в среде Darkstar, видимая архитектура складывается из набора служб в стеке. Набор служб может изменяться и настраиваться для конкретной конфигурации, но в нем всегда присутствуют четыре базовых службы, образующих ядро рабочей среды (рис. 3.2).

Основные службы

Важнейшей из служб уровня стека является служба данных, которая используется для хранения, выборки и обработки всех долгосрочных данных игры или виртуального мира. Термин «долгосрочность» здесь понимается шире, чем в других системах. В играх или виртуальных

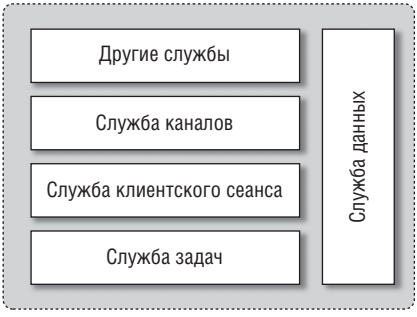


Рис. 3.2. *Стек Darkstar*

мирах, написанных для среды Darkstar, долгосрочными считаются любые данные, существование которых не ограничивается рамками одной задачи. Для их хранения должна использоваться служба данных. Напомню, что программная модель предполагает (и требует!) кратковременность задач, так что почти все данные, обеспечивающие представление игры или мира на стороне сервера, будут долгосрочными. Служба данных также связывает воедино разные копии игры или мира, работающие на разных серверах, так как все эти копии совместно используют один (концептуальный) экземпляр службы данных. Все копии обладают доступом к одинаковым данным и все копии могут читать и изменять данные, хранение которых обеспечивается этой службой.

На первый взгляд для реализации службы было бы естественно использовать базу данных, но в действительности требования к хранению сильно отличаются от тех, которые обычно обусловливают применение баз данных. Количество статических отношений между объектами в хранилище минимально, а сложные запросы к содержимому хранилища в игре обычно не нужны. Достаточно простой схемы назначения имен со ссылками на объекты, существующими на уровне языка программирования. Хранилище данных также должно быть оптимизировано для снижения задержки, а не для повышения пропускной способности. Количество объектов, с которыми работает любая отдельная задача, обычно невелико (согласно нашим предварительным оценкам на прототипах игр и миров – около дюжины объектов на задачу), причем примерно половина объектов, с которыми работает задача, изменяется в процессе выполнения.

Вторая служба стекового уровня – служба задач – используется для планирования и выполнения задач, сгенерированных либо в ответ на событие, полученное от клиентов, либо внутренней логикой сервера

игры или мира. Большинство задач относится к категории одноразовых, сгенерированных вследствие действия клиента; они получают информацию от службы данных, обрабатывают данные (возможно, с дополнительной передачей результата), после чего завершаются. Задачи также могут генерировать другие задачи или могут генерироваться как периодические задачи, выполняемые в определенные моменты или через определенные интервалы времени. Все задачи должны иметь короткий срок жизни; максимальное время существования задачи настраивается в конфигурации, но по умолчанию оно ограничивается 100 миллисекундами.

Программист игры или виртуального мира имеет дело с одной задачей, сгенерированной по событию или логикой самого сервера, но во внутренней реализации инфраструктура Darkstar планирует максимально возможное количество одновременно выполняемых задач. В частности, задачи, генерируемые серверной логикой, выполняются параллельно с задачами, сгенерированными по клиентским событиям.

Такое параллельное выполнение создает риск конфликтов доступа к данным. Для предотвращения упомянутых конфликтов служба задач должна взаимодействовать со службой данных. Во внутренней реализации незаметно для программиста серверной части каждая задача, планируемая службой задач, «упаковывается» в транзакцию. Транзакция гарантирует, что либо все операции задачи будут выполнены как одно целое, либо не будет выполнена ни одна из них. Кроме того, любые попытки изменения значений объектов, хранящихся в службе данных, проходят через эту службу. Если сразу несколько задач пытаются изменить один объект данных, то все эти задачи, кроме одной, отменяются и планируются заново для выполнения в будущем. Оставшаяся задача выполняется до завершения. После ее выполнения могут активизироваться другие задачи. Хотя серверный программист может указать, что данные, с которыми он обращается, могут быть изменены, это необязательно. Если объект данных просто читается, а изменяется позднее, то служба данных обнаружит факт изменения перед закреплением транзакции. Обозначение предполагаемого изменения в момент чтения – оптимизация, упрощающая раннее выявление конфликтов, но даже если вы не сообщите заранее о намерениях изменить данные, это не повлияет на правильность работы программы.

Упаковка задач в транзакции означает, что коммуникационные механизмы тоже должны быть транзакционными, а сообщения должны отправляться только при закреплении транзакции, включающей задачу-отправителя. Для решения этой задачи используются две оставшиеся основные службы стека Darkstar.

Коммуникационные службы

Первая из этих служб – служба сеансов – выполняет функции посредника между клиентом и сервером игры/виртуального мира. При входе и прохождении аутентификации между клиентом и сервером создается сеанс. Серверы получают сообщения, отправляемые клиентом во время сеанса, и разбирают содержимое сообщений для определения задач, которые необходимо сгенерировать для полученных сообщений. Клиенты ожидают получения ответов от сервера на этом канале. Сеансы скрывают реальные конечные точки канала связи от клиента и сервера – этот фактор играет важную роль в стратегии масштабирования Darkstar. Сеанс также отвечает за соблюдение правильного порядка сообщений. Сообщение от клиента не будет доставлено, если задачи, сгенерированные в результате доставки предыдущего сообщения, еще не были завершены. Такое упорядочение задач службой сеансов значительно упрощает службу задач, которая может считать, что все задачи в любой конкретный момент времени действительно параллельны. Упорядочение сообщений от конкретного клиента – единственная гарантия определенного порядка сообщений в инфраструктуре Darkstar; с точки зрения внешнего наблюдателя порядок поступления сообщений от разных клиентов может сильно отличаться от порядка, воспринимаемого игрой или виртуальным миром.

Вторая коммуникационная служба, всегда присутствующая в стеке Darkstar, – служба каналов. Каналы относятся к типу коммуникаций «один ко многим». На концептуальном уровне к каналу может присоединиться любое количество клиентов, а любое передаваемое по каналу сообщение доставляется всем подключенным к нему клиентам. На первый взгляд кажется, что ситуация идеально подходит для технологий одноранговых коммуникаций, которые позволяют клиентам напрямую взаимодействовать друг с другом, не создавая нагрузки на сервер. Однако такие взаимодействия должны контролироваться кодом, который следит за тем, чтобы клиенты не обменивались недопустимыми сообщениями и не мошенничали за счет использования разных клиентских реализаций. Предполагается, что клиент находится под контролем пользователя или игрока, поэтому доверять коду на клиенте нельзя – исходный клиентский код легко подменяется другой, «адаптированной» версией. Следовательно, все сообщения каналов все равно должны проходить через сервер, а их «благонадежность» должна проверяться серверной логикой.

Одна из сложностей, связанных с использованием каналов и сеансов, заключается в том, что они должны соблюдать транзакционную семантику задач. Таким образом, фактическая передача сообщения по

сеансовой линии или каналу не может происходить при вызове соответствующего метода `send()`; она может происходить только при закреплении транзакции той задачи, в которой вызывается этот метод.

С реализацией этих коммуникационных механизмов в нашем распоряжении появляются некоторые компоненты, необходимые для второй части механизма масштабирования. Поскольку все коммуникации должны проходить через абстракции сеансов или каналов Darkstar и поскольку эти абстракции не предоставляют клиенту или серверу доступ к фактическим конечным точкам обмена данными, между взаимодействующими сущностями и фактическим началом/концом передачи данных появляется абстрактная прослойка. Это означает, что конечную точку коммуникаций с сервером можно переместить с одного компьютера системы Darkstar на другой, не влияя на представление клиента о передаче данных. С точки зрения клиента все взаимодействия происходят в конкретном сеансе или канале. С точки зрения игры или логики виртуального мира взаимодействия также осуществляются через один сеанс или канал. Однако инфраструктура нижнего уровня может переместить сеанс или канал с одного компьютера на другой, чтобы обеспечить перераспределение нагрузки по мере ее изменения со временем.

Портируемость задач

Чтобы распределение нагрузки стало возможным при выбранной нами программной модели и используемых основных службах стека, задачи, выполняемые в ответ на сгенерированные клиентом или игрой события должны быть совместимы между всеми машинами, на которых работают копии логики игры или мира в стеке Darkstar. Сами задачи пишутся на Java[1], а это означает, что они могут выполняться на любых других компьютерах, у которых в стек выполнения входит та же версия виртуальной машины Java. Все данные, читаемые и обрабатываемые задачей, должны получаться от службы данных, общей для всех экземпляров игры или виртуального мира, и стеков Darkstar на всех компьютерах. Посредником в обмене данных является служба сеансов или каналов, которая абстрагирует конечные точки коммуникаций и позволяет переместить любой сеанс или канал с одного сервера

[1] А точнее, все задачи состоят из байт-кода, который может выполняться виртуальной машиной Java. Выбор языка уровня исходного кода ни на что не влияет; для нас важно только то, чтобы откомпилированная форма могла выполняться на любом компьютере из распределенного набора, обеспечивающего работу игры или виртуального мира.

на другой. Таким образом, любая задача может выполняться на любых экземплярах игрового сервера без изменения семантики задачи.

Благодаря этому появляется возможность сделать базовый механизм масштабирования Darkstar простым на вид. Если нагрузка на компьютер становится слишком высокой, некоторые задачи с него перемещаются на другой, менее загруженный. Если перегружены все компьютеры, то в группу добавляется новый компьютер с копией сервера игры или виртуального мира поверх стека Darkstar, а подсистема распределения нагрузки начинает передавать задачи на новый компьютер.

Отслеживание нагрузки на отдельных компьютерах и ее перераспределение в случае необходимости является задачей метаслужб. Это службы сетевого уровня, которые остаются невидимыми для программиста игры или виртуального мира; с другой стороны, они видны для служб стека Darkstar (и сами могут наблюдать за ними). Например, метаслужбы следят за тем, какие компьютеры работают в текущий момент (и не произошли ли на этих компьютерах сбои), какие пользователи связаны с задачами конкретного компьютера, а также отслеживают текущую нагрузку на разные машины. Поскольку метаслужбы невидимы для программиста игры или виртуального мира, их можно в любой момент изменить без последствий для правильности игровой логики. Это позволяет нам экспериментировать с разными стратегиями динамического распределения нагрузки в системе, а также расширять набор метаслужб, необходимых для инфраструктуры.

Механизм, используемый нами для масштабирования, также обеспечивает высокую степень отказоустойчивости в системе. Учитывая машинно-независимую природу данных, используемых задачами и коммуникационными механизмами, становится ясно, что задачу можно перемещать с одного компьютера на другой. Но если на компьютере произойдет сбой, как восстановить задачи, которые на нем выполнялись? Ответ: сами задачи представляют собой долгосрочные объекты, сохраняемые службой данных для всей системы. Таким образом, если на компьютере происходит сбой, то все выполнявшиеся на нем задачи рассматриваются как отмененные транзакции и перебрасываются на другие компьютеры. Хотя задержка от такой перепланировки может превышать задержку от перепланировки отмененной транзакции на том же компьютере, система все равно будет работать правильно. В худшем случае пользователь системы (игрок или обитатель виртуального мира) заметит небольшую задержку в реакции системы. Задержки могут раздражать, но в гораздо меньшей степени, чем традиционные последствия сбоя сервера в игре или виртуальном мире, когда игроку как минимум приходится подключаться заново, да еще и с возможностью потери существенного игрового состояния.

Размышления об архитектуре

Вероятно, первый вопрос, который вы зададите об архитектуре и ее реализации, – насколько эффективно она работает? Хотя преждевременная оптимизация архитектур порождает множество грехов, ничто не мешает спроектировать архитектуру, которая в любом варианте реализации будет работать неудовлетворительно. Вследствие одного из основных решений архитектуры Darkstar это беспокойство выглядит обоснованно. А из-за особенностей игровой отрасли определить производительность серверной инфраструктуры довольно сложно.

Трудности с определением производительности инфраструктуры сервера игры или виртуального мира обусловлены простым фактом: для крупномасштабных MMO или виртуальных миров не существует эталонных тестов или общепринятых образцов. Отсутствие контрольных тестов не удивительно, так как серверные компоненты большинства игр или виртуальных миров строятся для конкретной игры или виртуального мира «с нуля». Существует лишь несколько обобщенных инфраструктур, которые могут использоваться в качестве стандартных структурных элементов; как правило, они извлекаются из существующей игры или мира после завершения работы над системой и передаются тем, кто строит аналогичные игры. Не существует общепринятых эталонных тестов для тестирования новых или сравнения существующих инфраструктур – как среди относительно новых явлений игровой отрасли, так и среди исторически сформировавшихся технологий индустрии развлечений.

Также практически отсутствует информация о предполагаемых нагрузках на сервер игры или виртуального мира (вычисления, обработка данных, коммуникации), на основании которой можно было бы построить эталонные тесты или тесты производительности. Отчасти это объясняется нестандартной природой создаваемых серверов. Каждый сервер строится под конкретную игру или виртуальный мир, а, следовательно, настраивается под конкретные характеристики рабочей нагрузки этой игры или мира. В еще большей степени проблема обусловлена скрытностью игровой отрасли, в которой любые сведения о находящихся в разработке играх ревностно охраняются, а информация о реализации готовых игр, во-первых, так же тщательно охраняется, а, во-вторых, многие считают ее неинтересной. Графическому оформлению, сюжетам и новшествам игрового процесса уделяется гораздо больше внимания, чем архитектуре игрового сервера или механизмам масштабирования игры для текущего количества игроков (кстати, эта статистика тоже часто скрывается). Таким образом, даже получить

информацию о нагрузке на серверы, создаваемой современными играми и виртуальными мирами, весьма непросто.

Даже когда нам удавалось расспросить разработчиков о том, какую нагрузку на сервер создает их игра или виртуальный мир, сведения часто оказывались неправильными. Дело вовсе не в том, что они пытались сохранить какие-то коммерческие преимущества, дезинформируя нас о том, что реально происходит на серверах, – они просто сами не знали. На игровых серверах практически отсутствуют средства сбора информации о том, что делает сервер и насколько эффективно он работает. Анализ работы таких серверов в лучшем случае становится занятием эмпирическим. Программист работает над сервером до тех пор, пока игровой процесс не войдет в рамки приемлемой производительности, но эта задача решается итеративным методом, а не тщательными измерениями параметров кода. В таких системах ремесла намного больше, чем науки.

Я вовсе не хочу сказать, что серверы, обеспечивающие работу таких игр и виртуальных миров, недостаточно хорошо спроектированы или построены. Часто это настоящие шедевры программной мысли и эффективности, демонстрирующие преимущества специализированных серверов для требовательных приложений. Однако практика построения нового сервера для каждой игры или мира означает, что разработка не приводит к накоплению знаний относительно того, что необходимо для таких серверов, и не существует общепринятого механизма сравнения таких инфраструктур.

Параллелизм и задержка

Отсутствие информации о том, что необходимо для приемлемой производительности сервера, вызывало особенную озабоченность у группы Darkstar, потому что некоторые из принимаемых нами решений противоречили традиционным воззрениям относительно того, как добиться хорошей производительности от сервера игры или виртуального мира. Возможно, самое радикальное отличие архитектуры Darkstar от традиционных решений заключалось в отказе от хранения сколько-нибудь значимой информации в основной памяти сервера. Требование о долгосрочном хранении в службе данных всей информации, выходящей за рамки конкретной задачи, занимает центральное место в функциональности Darkstar. Оно позволяет инфраструктуре выявлять проблемы параллелизма, что в свою очередь позволяет системе скрыть эти проблемы от программиста, но при этом сохранить за сервером все преимущества многоядерных архитектур. Кроме того, оно является ключевым компонентом общей схемы масштабирования, так как задачи

могут перемещаться между компьютерами для равномерного распределения нагрузки в группе компьютеров.

Долгосрочное хранение состояния считается ересью в мире серверов игр и виртуальных миров, где фактор задержки выходит на первый план. Традиционно считалось, что вся информация должна храниться лишь в основной памяти – только в этом случае задержка будет приемлемо низкой, а сервер обеспечит требуемое время отклика. Возможно, система может время от времени сохранять «снимки» состояния, но необходимость в интерактивных скоростях означает, что все долгосрочные операции должны выполняться редко и в фоновом режиме. Казалось бы, наша архитектура основана на предпосылке, которая заведомо не позволит ей обеспечить приемлемую производительность для предполагаемой аудитории.

Безусловно, долгосрочное хранение данных оказывает серьезное влияние на архитектуру, а обращение к информации через службу данных вводит в архитектуру значительную задержку. И все же мы полагаем, что выбранный подход более чем конкурентоспособен; на это есть несколько причин. Во-первых, мы считали, что различие между обращением к данным в основной памяти и обращением через службу данных можно сделать намного меньше, чем традиционно считалось. Хотя концептуально каждый объект с жизненным циклом, выходящим за рамки одной задачи, должен читаться и записываться в долгосрочное хранилище, реализация такого хранилища может использовать многолетний опыт кэширования баз данных, чтобы свести к минимуму задержки обращения к данным.

Это особенно относится к случаям, когда доступ к набору объектов локализуется на определенном сервере. Если все задачи, использующие определенный набор объектов, выполняются на одном сервере, то кэширование обеспечит доступ к объектам практически за такое же время, как при чтении/записи в основной памяти (со всеми действующими ограничениями по жизнеспособности данных). Задачи могут связываться с конкретными игроками или пользователями виртуального мира. И здесь требование о том, чтобы доступ к данным и коммуникации осуществлялись только через службы, предоставленные инфраструктурой, позволяет нам организовать сбор информации о закономерностях обращений к данным и коммуникаций в игре или мире за некоторый период времени. Располагая такой информацией, можно очень точно оценить оптимальную группировку игроков. Имея возможность перемещать игроков на другие серверы по своему усмотрению, мы можем активно формировать их оптимальную группировку на основании отслеживаемого поведения. Это позволит применить стандартные методы кэширования, хорошо известные в мире баз дан-

ных, для сокращения задержек доступа и хранения долгосрочной информации.

На первый взгляд, эта идея сильно напоминает географическую декомпозицию, которая в настоящее время применяется для реализации масштабирования в крупномасштабных играх и виртуальных мирах. Разработчики сервера разбивают мир на области, которые связываются с определенными серверами, и группировка игроков осуществляется по областям. Игроки, находящиеся в одной области, с большей вероятностью будут взаимодействовать друг с другом, нежели с обитателями других областей, а это повышает эффективность их группировки на сервере. Различие заключается в том, что географические декомпозиции осуществляются в ходе разработки игры, а затем материализуются в исходном коде на сервере. Наш метод группировки основан на информации, полученной во время выполнения; он может динамически настраиваться в соответствии с закономерностями игрового процесса или взаимодействий в ходе самой игры. Разница примерно такая же, как между оптимизациями на стадии компиляции и оптимизациями JIT (Just-In-Time). Первые направлены на все возможные запуски программы, а вторые – на конкретный текущий запуск.

Вряд ли нам удастся полностью нивелировать различия между доступом к основной памяти и долгосрочному хранилищу, но мы и не думаем, что нам так уж необходимо превзойти производительность инфраструктур, работающих с основной памятью. Вспомните, что долгосрочное хранение всех данных позволяет нам использовать многопоточную модель (а, следовательно, возможности многоядерного процессора) на сервере. Конечно, параллелизм не будет идеальным (то есть использование каждого дополнительного ядра будет 100-процентным), но мы уверены (и предварительные результаты подкрепляют эту уверенность), что из параллелизма в играх и виртуальных мирах можно извлечь значительную пользу. Если выгода от использования параллелизма превысит величину вводимой задержки, то общая производительность игры или виртуального мира только повысится.

Ставка на будущее

Ориентируясь на многопоточность на многоядерных процессорах, мы фактически делаем ставку на путь развития процессоров в ближайшем будущем. В настоящее время серверы строятся на базе процессоров, содержащих от 2 до 32 ядер; мы полагаем, что архитектура процессоров будет развиваться по пути увеличения количества ядер, а не повышения их тактовой частоты. Когда несколько лет назад мы начинали работу над проектом, эта ставка казалась куда более гипотетической, чем сейчас. В то время мы представляли свои разработки как упражне-

ния «что если», говоря, что мы экспериментируем с архитектурой, которая могла бы воплотиться в жизнь, если бы производительность процессоров повышалась по пути увеличения числа поддерживаемых потоков, а не по пути роста тактовой частоты. Это одно из преимуществ выполнения подобных проектов в исследовательских лабораториях: мы можем выбрать значительно более рискованное решение для исследования области, которая может оказаться коммерчески рентабельной. При современных тенденциях технологий процессоров архитектурные решения, основанные на многопоточности, выглядят намного перспективнее, чем на момент принятия этого решения[1].

Даже если параллелизм будет использоваться всего на 50% своего потенциала, мы всего равно достигнем «точки безубыточности», даже если доступ к долгосрочному хранилищу будет осуществляться гораздо медленнее (от 2 до 16 раз) доступа к основной памяти. Мы предполагаем, что нам удастся добиться успехов и в параллелизме, и в сокращении различий между обращением к долгосрочному хранилищу и основной памятью. Но многое будет зависеть от закономерностей их использования, основанных на инфраструктуре (которые, как упоминалось ранее, довольно трудно выявить).

Мы также не считаем сокращение задержки единственной целью своей инфраструктуры. Храня все состояние игры или объектов мира в хранилище данных, мы сводим к минимуму потери данных в случае сбоя сервера. Действительно, в большинстве случаев сбой сервера проявится только в виде непродолжительного повышения задержки на время перемещения задач (представленных долгосрочными объектами) со сбойного сервера на альтернативный; потерь данных при этом быть не должно. Некоторые схемы кэширования могут привести к потере нескольких секунд игры – но даже это лучше, чем схемы с периодическим сохранением «снимков», являющиеся основной формой сохранения данных в современных онлайновых играх и виртуальных мирах. В таких инфраструктурах сбой сервера в неподходящий момент может привести к потере нескольких часов игры. Если задержки остаются приемлемыми, повышенная надежность механизма долгосрочного хранения данных Darkstar открывает преимущества как для разработчиков системы, построенной на базе подобной инфраструктуры, так и для ее пользователей.

---

[1] Что лишний раз доказывает старую истину: на ранних стадиях проектирования важнее всего обычное везение.

Упрощение работы программиста

В самом деле, если бы единственной целью разработчика было сокращение задержек при возможном масштабировании, то ему следовало бы написать собственную распределенную многопоточную инфраструктуру, адаптированную для конкретной игры. Но для этого разработчику придется иметь дело со всеми сложностями распределенного и многопоточного программирования. Прежде чем зацикливаться на скорости, стоит вспомнить, что второй (но не менее важной!) целью Darkstar является возможность создания многопоточных распределенных игровых миров, при которой программист работает с моделью программирования на одном компьютере с одним потоком.

В значительной степени мы с этой целью справились. Благодаря упаковке всех задач в транзакции и выявлению конфликтов доступа на уровне службы данных программист пользуется преимуществами многопоточной модели, обходясь без протоколов блокировки, синхронизации или семафоров. Ему не нужно беспокоиться о том, как перевести игрока на другой сервер, потому что Darkstar автоматически распределяет нагрузку. Первые участники сообщества разработчиков сочли, что программная модель при всех ее ограничениях естественно подходит для создаваемых ими игр и виртуальных миров.

К сожалению, выяснилось, что скрыть от программиста абсолютно все не удастся. Это стало очевидно, когда первая игра, написанная на основе Darkstar, проявляла минимальный параллелизм (и работала крайне медленно). Анализ исходного кода довольно быстро выявил причину. Структуры данных игры были написаны так, что любое изменение состояния в игре затрагивало один центральный объект, который координировал все выполняемые операции. Использование центрального объекта приводит к тому, что все действия в игре выполняются последовательно, а инфраструктура не может обнаружить возможности параллельной обработки или использовать их.

Когда это выяснилось, мы долго обсуждали с разработчиками игр необходимость планирования параллельного доступа при проектировании объектов. Анализ объектов данных игры выявил ряд сходных случаев, в которых параллелизм был (непреднамеренно) подавлен из-за решений, принятых в ходе проектирования данных. Как только объекты были переработаны, производительность системы возросла более чем на порядок.

Из этого следовал вывод, что разработчики, использующие архитектуру Darkstar, не могут полностью изолироваться от параллельной и распределенной природы системы. Тем не менее их соприкосновение с соответствующими сторонами системы обходится без обычных проблем

управления доступом, блокировки и обеспечения взаимодействий между распределенными частями системы. Вместо этого они должны проследить за тем, чтобы объекты данных определялись с расчетом на максимальную эффективность параллельного доступа. В более общем смысле это обычно означает, что определяемые объекты автономны, а их операции не зависят от состояния других объектов (что само по себе считается хорошим архитектурным принципом в любой системе).

В архитектуре Darkstar осталось еще много такого, что мы не протестировали или не поняли в полной мере. Хотя мы построили систему, позволявшую развернуть игру или виртуальный мир на нескольких компьютерах прозрачно (в основном) для программиста сервера, еще не протестирована возможность добавления других служб, кроме базовых. С учетом транзакционной природы задач Darkstar это может оказаться сложнее, чем казалось на первый взгляд. Мы надеемся, что дополнительные службы смогут обойтись без участия в транзакциях основных служб. Мы также только приступили к экспериментам с разными способами сбора информации о нагрузке на систему и ее распределении. К счастью, механизмы управления нагрузкой полностью скрыты от программиста, использующего систему, и это позволяет нам пользоваться старыми методами или вводить новые без последствий для пользователей Darkstar.

В архитектуре Darkstar представлен целый ряд новаторских методов, благодаря которым она заслуживает внимания. Это одна из немногочисленных попыток построить инфраструктуру игры или виртуального мира, не уступающую крупным коммерческим решениям по надежности и безотказности, но при этом удовлетворяющую требованиям к задержке, коммуникациям и масштабированию, характерным для игровой отрасли. Стремясь к повышению эффективности за счет использования дополнительных компьютеров и программных потоков, мы надеемся скомпенсировать возрастание задержки за счет применения механизмов долгосрочного хранения данных. Наконец, специфика мира игр и виртуальных миров, с толстыми клиентами и тонкими серверами, сильно отличается от типичных условий построения распределенных систем с высокой степенью параллелизма. Пока рано судить, насколько успешной будет эта архитектура, но мы полагаем, что даже в своем нынешнем состоянии она уже интересна.

	Принципы и свойства		Структуры
	Гибкость	✓	Модуль
✓	Концептуальная целостность	✓	Зависимость
✓	Возможность независимого изменения		Обработка
✓	Автоматическое распространение		Доступ к данным
✓	Удобство построения		
	Адаптация к росту		
	Сопротивление энтропии		

4

Фото на память

Майкл Найгард

С момента появления первых ферротипов и дагерротипов фотография всегда кажется нам чем-то особенным, даже волшебным. Фотография сохраняет мимолетное мгновение лучше, чем это делает наша ненадежная память. Но хорошие портреты не просто сохраняют момент, а еще и освещают его. Они запечатлевают взгляд, выражение или характерную позу, в которых проявляется наша индивидуальность.

Всем родителям детей, учащихся в школах США, хорошо знакомо название Lifetouch. Эта фирма ежегодно фотографирует большинство учеников начальных и средних школ. Однако не всем известно, что у Lifetouch также имеются студии художественной портретной съемки. Lifetouch Portrait Studios (LPS) работает в крупных магазинах по всей стране наряду с сетью студий «Flash!» в торговых центрах. В этих студиях фотографы LPS снимают портреты, которые после этого живут еще много лет.

Цифровая фотография изменила всю фотографическую отрасль, и фирма LPS не стала исключением. Огромные ролики пленки, камеры на штативах исчезают, а им на смену приходят профессиональные цифровые зеркальные камеры с картами памяти. Фотограф освобождается

от громоздкой аппаратуры, он может ходить по студии, пробовать разные углы и подходить к объекту съемки ближе, чем когда-либо. Короче говоря, у фотографа появляется больше свободы для создания замечательных снимков. Работая с цифровой камерой, он преобразует фотоны в электроны, но рано или поздно некоторая система должна снова превратить эти электроны в атомы бумаги и краски.

В 2005 году я и мои коллеги из фирмы Advanced Technologies Integration (ATI) в Миннеаполисе работали вместе с разработчиками LPS над новой системой, которая должна была решать именно эту задачу.

Возможности и ограничения

Архитектура системы зависит от двух движущих сил. Что должна делать система? В рамках каких ограничений она должна работать? Эти вопросы определяют пространство задачи.

Мы создаем (и одновременно исследуем) пространство задачи, анализируя эти движущие силы, перемещаясь между положительным полюсом требуемого поведения системы и отрицательным полюсом ограничений. Иногда нам удается создать элегантное, даже красивое решение, когда реакции на отдельные ограничения складываются в единое целое. С гордостью скажу, что проект Creation Center был именно таким.

В этом проекте мы столкнулись с некоторыми непререкаемыми фактами. Одни вытекали непосредственно из природы бизнеса; другие могли изменяться, но вне пределов нашего контроля. И те, и другие рассматривались как объективные внешние условия. Эти факты перечислены в левом столбце на рис. 4.1.

Разные варианты оформления

LPS сейчас поддерживает разные стандартные варианты оформления (бренды), и в будущем к ним могут добавиться новые. Как минимум, Creation Center должен поддерживать два визуально различающихся скина, причем добавление новых скинов должно обходиться без значительных усилий.

Пользователи – фотографы, а не специалисты по компьютерной графике

Фотографы обучены работать с камерой, а не с Photoshop. Когда неопытный пользователь садится за Photoshop, скорее всего, результат будет выглядеть убого. Это мощный инструмент для опытных пользователей, и фотографу в студии незачем тратить силы на освоение сложных программ. Photoshop и его аналоги также замедляют технологический процесс. Работники студий должны иметь возможность быстро создать качественное изображение.

Рис. 4.1. *Факты, движущие силы и аспекты архитектуры Creation Center*

Студии разбросаны на больших расстояниях

Студии географически удалены друг от друга, а возможности местной технической поддержки минимальны. Доставка или замена оборудования потребует пересылки компонентов.

В сетях возможны сбои

Некоторые студии не имеют сетевых подключений. Но даже если подключение имеется, студия не должна прерывать работу, если сеть вдруг выйдет из строя.

Персонализация фотографий

Клиент сам выбирает дизайн и текст для оформления фотографии.

Производство централизовано

Высококачественные фотопринтеры получают все более широкое распространение, но для изготовления фотопродуктов, которые будут существовать десятилетиями, необходимо более дорогостоящее оборудование.

Производительность играет важную роль

Принтеры также являются «узким местом» производственного процесса. Следовательно, все остальные этапы процесса должны подчиняться этим ограничениям.

Перечисленные факты определяют движущие силы, которые необходимо сбалансировать в архитектуре. Движущие силы проекта часто воспринимаются как фундаментальные, но на самом деле они таковыми не являются. Они формируются на основе контекста, в котором существует система. При изменении контекста движущие силы теряют значимость или даже меняют полярность.

Для удовлетворения этих требований мы выбрали соответствующие архитектурные конструкции; они показаны в правом столбце рис. 4.1. Конечно, у Creation Center есть и другие аспекты архитектуры, заслуживающие внимания, однако эти представляют особый интерес. Кроме того, как мне кажется, они также демонстрируют разделение сфер влияния в системе.

Но прежде чем переходить к углубленному изучению конкретных функций, необходимо познакомиться с еще одной составляющей контекста – технологическим процессом системы.

Технологический процесс

Типичная студия содержит от двух до четырех фотопавильонов с профессиональным освещением, фоном и реквизитом. Фотограф производит съемку в павильоне. За его пределами фотографы обслуживают клиентов, назначают время посещения и выдают готовые заказы.

После завершения сеанса съемки фотограф садится за любую из нескольких рабочих станций, чтобы загрузить фотографии с карты памяти. Он удаляет явно неудачные фотографии – с закрытыми глазами, кислым выражением лица, отвернувшимися детьми и т. д. Фотографии, оставшиеся после удаления, называются «базовыми». Далее фотограф вносит в них улучшения: от простых тональных преобразований (черно-белые фотографии, сепия) до сложных коллажей из нескольких фотографий. Например, фотограф делает групповой портрет трех детей и вставляет его в дизайн с тремя «рамками» для портрета каждого ребенка.

Завершив доработку, фотограф помогает клиенту выбрать размеры и комбинации фотографий для печати, от портретов 20×25 см до «карманных» карточек. Также возможна крупноформатная печать. Клиент может заказать портрет размером до 63×76 см, чтобы вставить его в раму и повесить на стену.

После завершения обработки заказа фотограф переходит к следующему сеансу.

В конце рабочего дня директор студии записывает DVD с сегодняшними заказами и отправляет их в центр печати.

В центр печати ежедневно приходят сотни дисков (об их содержимом мы поговорим чуть позже). На DVD хранятся заказы и фотографии, которые необходимо напечатать и отослать обратно в студию, где их заберет клиент. Но перед выводом на печать фотографии с итоговым разрешением печати необходимо преобразовать в графическую форму. Изображения, готовые к печати, имеют огромные размеры. Портрет 63×76 см в высоком разрешении для качественной печати содержит более 100 миллионов пикселей, каждый из которых представлен 32-разрядным цветовым значением. Каждый пиксель формируется в соответствии с дизайном, выбранным фотографом в студии. В зависимости от композиции конвейер визуализации может содержать от 6 до 10 этапов. Простое построение изображения занимает от 2 до 5 минут, но на сложные композиции для крупноформатных снимков уходит по 10 минут и более.

В то же время принтер выдает несколько готовых отпечатков в минуту. За постоянную занятость принтеров отвечает PCS (Production Control System) – сложная система, которая планирует работу и управляет системой визуализации, обеспечивает хранение графических данных и ставит задания в очередь печати.

Когда выполненный заказ возвращается в студию, директор сообщает заказчику, что тот может прийти за своим заказом.

Этот технологический процесс отчасти обусловлен бизнес-контекстом LPS, отчасти – нашим выбором логического разбиения системы на компоненты. А теперь давайте рассмотрим другие аспекты архитектуры, представленные на рис. 4.1.

Архитектурные грани

Попытка свернуть структуру многомерной динамической системы в форму линейного повествования всегда создает проблемы – как при изложении нашего видения планируемой системы, так и при попытке объяснить взаимодействие частей в уже существующей системе. Гипертекст может упростить «описание слона слепцами», но бумага пока не очень хорошо поддерживает гиперссылки.

Разглядывая каждую из этих «граней», помните, что они всего лишь являются разными представлениями общей системы. Например, мы использовали модульную архитектуру для поддержки разных сценариев

развертывания. В то же время каждый модуль строится в иерархической архитектуре. Эти представления ортогональны, но между ними существует пересечение. Каждый набор модулей соответствует одной иерархической структуре и каждый уровень иерархии проходит через все модули. Мы были горды тем, что нам удалось обеспечить разделение этих сфер влияния, но при этом обеспечить их взаимоподдержку.

Модули и программа Launcher

Все это время мы думали не о «приложении», а о «семействе продуктов», потому что нам нужно было обеспечить поддержку нескольких сценариев развертывания для одного базового кода. В частности, с самого начала были известны следующие конфигурации:

Студийный клиент

В студии устанавливается от 2 до 4 рабочих станций. Фотографы используют их на всех этапах технологического процесса, от загрузки графики до создания заказов.

Студийный сервер

Центральный сервер каждой студии использует MySQL для работы со структурированными данными (сведения о посетителях, заказы и т. д.). Сервер должен обеспечивать намного более надежное хранение данных, чем рабочие станции, с применением защищенных массивов RAID. На студийном сервере также происходит запись дневных заказов на DVD.

Ядро визуализации

Мы также решили построить собственное ядро визуализации для печати. Если один и тот же код выводит графику на экран в студии и преобразует ее в графический формат, готовый к печати, то мы можем быть абсолютно уверены в том, что клиент получит именно то, что хотел.

Сначала мы полагали, что разные конфигурации будут представлять собой разные наборы файлов *.jar*. Мы создали несколько каталогов верхнего уровня для хранения кода каждого развертывания, а также общую папку *Common*. В каждой папке верхнего уровня находились собственные каталоги *source*, *test* и *bin*.

Прошло совсем немного времени, и мы разочаровались в этой структуре. Во-первых, в иерархии был один гигантский каталог */lib*, в котором постепенно начали накапливаться библиотеки времени сборки и времени выполнения. Проблемы возникли и с размещением ресурсов: графики, цветовых профилей, конфигураций Hibernate, тестовых изображений и т. д. Некоторых участников также раздражало то, что

нам приходилось управлять зависимостями файлов *.jar* вручную. Тогда, на заре проекта, целые пакеты запросто могли оказаться в постороннем каталоге. А во время выполнения класс отказывался загружаться, потому что он зависел от классов, упакованных в другой файл *.jar*.

Перелом наступил тогда, когда после трех итераций в проект была введена поддержка Spring[1]. Мы следовали методологии «гибкой архитектуры»: поддерживали минимальную архитектуру и добавляли новые архитектурные возможности только тогда, когда затраты от их отсутствия превышали затраты на их реализацию. Это то, что в методологии Lean называется «последним ответственным моментом». Поначалу мы лишь в общих чертах представляли себе инфраструктуру Spring, поэтому решили не зависеть от нее, хотя и ожидали, что позднее она нам понадобится.

При добавлении Spring к проблемам зависимостей файлов *.jar* добавились проблемы с конфигурационными файлами. Каждая конфигурация развертывания должна была иметь собственный файл *beans.xml*, но более половины компонентов дублировалось между файлами – очевидное нарушение принципа «не повторяйся[2]», и несомненный источник дефектов. Нельзя полагаться на ручную синхронизацию определений в XML-файлах под тысячу строк. Кроме того, разве XML-файл размером под тысячу строк не является тревожным признаком?

Нам требовалось решение, которое обеспечивало бы модульное разбиение файлов *beans*, управляло зависимостями файлов *.jar*, позволяло хранить библиотеки поблизости от кода, в котором они используются, и управлять путем к классам во время построения и выполнения.

Контекст приложения

Изучение Spring напоминает исследование огромной незнакомой местности. Это NetHack[3] мира инфраструктур; авторы продумали все без исключения. Странствия по документации javadoc часто приносят великую награду. Мы открыли свою золотую жилу в тот момент, когда я наткнулся на класс «контекста приложения».

В любом приложении Spring центральное место занимает «фабрика bean-компонентов». Она позволяет искать объекты по имени, создавать

---

[1] *http://www.springframework.org/*

[2] См. «The Pragmatic Programmer», Andrew Hunt, David Thomas, Addison-Wesley Professional (Эндрю Хант и Дэвид Томас «Программист-прагматик», Лори, 2009).

[3] Классическая компьютерная игра, хорошо известная обилием секретов и возможностей исследования. – *Примеч. перев.*

их в случае надобности, а также внедрять конфигурации и ссылки на другие bean-компоненты. Короче, фабрика управляет объектами Java и их конфигурациями. Самая распространенная реализация фабрики читает XML-файлы.

Контекст приложения расширяет фабрику и наделяет ее важнейшей способностью создания цепочек вложенных контекстов в соответствии с паттерном «цепочка ответственности» (Design Patterns, Gamma et al. 1994).

Объект `ApplicationContext` предоставил нам именно то, что требовалось: механизм разбиения bean-компонентов на несколько файлов, с загрузкой каждого файла в отдельном контексте приложения.

Далее нужно было найти способ создания цепочки контекстов приложений, желательно без применения гигантских сценариев командного процессора.

Зависимости модулей

Рассматривая каждый каталог верхнего уровня как модуль, я решил, что было бы естественно хранить в каждом модуле его метаданные. В этом случае модуль мог бы просто объявить свои конфигурационные файлы и пути к классам вместе с объявлением других необходимых модулей.

Я выделил каждому модулю собственный файл манифеста. Например, манифест модуля `StudioClient` выглядит так:

```
Required-Components: Common StudioCommon
Class-Path: bin/classes/ lib/StudioClient.jar
Spring-Config: config/beans.xml config/screens.xml config/forms.xml
 config/navigation.xml
Purpose: Selling station. Workflow. User Interface. Load images. Burn DVDs.
```

Разумеется, этот формат является производным от формата манифестов *.jar*. Я решил, что будет удобно совместить наши традиционные представления о «файле манифеста» со знакомым форматом.

Обратите внимание: модуль использует четыре разных файла bean-компонентов. Дополнительным преимуществом стала возможность разбиения определений по функциям. Разбиение помогло «разгрузить» главные конфигурационные файлы и обеспечило хорошее разделение сфер влияния.

Наша группа отдавала предпочтение автоматизированной документации, поэтому мы включили в процесс построения несколько аналитических операций. Когда все зависимости модулей явно записаны в манифесте, построение отчета в ходе автоматизированной сборки выпол-

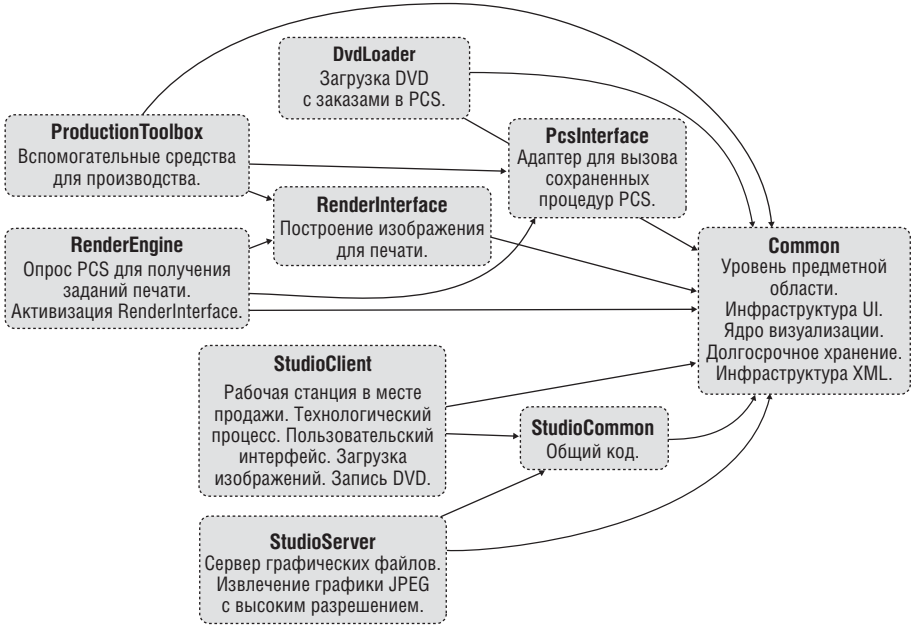


***Рис. 4.2.** Модули и зависимости*

няется тривиально. Немного разбора текста, передаем данные Graphviz – и получаем диаграмму зависимостей, изображенную на рис. 4.2.

С этими файлами манифестов нам необходим лишь механизм их разбора и выполнения полезных действий. Для этого я написал программу с неоригинальным названием Launcher.

Launcher

Я видел много настольных приложений Java, которые комплектуются громадными сценариями командного процессора для поиска JRE, настройки переменных окружения, построения пути к классам и т. д. Отвратительно!

Launcher получает имя модуля, разбирает файлы манифестов и строит транзитивное замыкание зависимостей модуля. Launcher следит за тем, чтобы модуль не включался дважды, и разрешает множество частичных упорядочений в полное упорядочение. На рис. 4.3 изображен полный граф зависимостей системы `StudioClient`. Эта система объявляет зависимости `StudioCommon` и `Common`, но Launcher включает только одну копию.

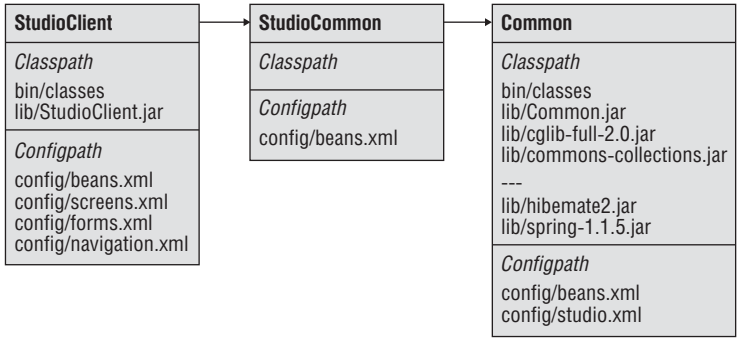


Рис. 4.3. *Разрешение зависимостей для StudioClient*

Чтобы избежать «загрязнения» путей к классам в управляющей среде (ANT на сборочной машине, JRE на рабочей станции), Launcher строит собственный загрузчик классов из объединенных путей. Все классы приложения загружаются этим загрузчиком, поэтому Launcher использует его для создания экземпляра инициализатора. Launcher передает конфигурационный путь инициализатору, который создает все объекты контекстов приложений. После конструирования контекстов приложение готово к работе.

В ходе работы над проектом мы несколько раз перерабатывали структуру модулей. Количество изменений в файлах манифестов и Launcher при этом было минимальным. В конечном итоге мы получили шесть сильно различающихся конфигураций развертывания, причем все они поддерживались одной структурой.

Модули имеют сходную структуру, но они не обязаны быть идентичными. Это было одним из побочных преимуществ нашего подхода. Каждый модуль может «спрятать» ту информацию, которая не представляет интереса для других модулей.

Графический интерфейс в стиле «киоск»

Студия нанимает работников за их умение обращаться с фотоаппаратом и общаться с посетителями (особенно с детьми), а не за компьютерные навыки. Возможно, дома они владеют Photoshop на профессиональном уровне, но в студии никто не ожидает от них подобного мастерства. Более того, в «горячую пору» студия может привлечь временных работников, поэтому очень важно, чтобы они как можно быстрее освоили работу с оборудованием.

А как же OSGi?

Когда мы начинали работу над этим проектом в конце 2004 года, инфраструктура OSGi только завоевывала популярность – главным образом благодаря тому, что она была принята в Eclipse. Мы в общих чертах познакомились с ней, но нас отпугнуло отсутствие широко доступной информации, опыта и методик.

Однако цели OSGi идеально подходили для тех задач, с которыми мы столкнулись. Поддержка разных конфигураций развертывания с общей кодовой базой, управление зависимостями между модулями, активизация их в правильной последовательности... Все те же проблемы.

Полагаю, тот факт, что мы не использовали OSGi, отчасти объяснялся неудачным выбором момента, а отчасти – нашим нежеланием идти на то, что мы воспринимали как дополнительный технический риск. Обычно я предпочитаю следовать принципу «бери и интегрируй» вместо принципа «сделай сам», но в данном случае меня отпугнула другая опасность: использование проектов с открытым кодом, известных минимальной поддержкой и слабыми сообществами, создает больший риск, чем хорошо понятные, общепринятые протоколы. Я также стараюсь избегать «псевдооткрытых» инфраструктур, которые в действительности правильнее было бы отнести к стандартам фирм-разработчиков. Обычно они служат интересам фирм-разработчиков, а не пользователей.

На тот момент нам было неясно, к какому лагерю будет относиться OSGi. Если бы мы занимались этим проектом сегодня, то, вероятно, воспользовались бы OSGi вместо разработки собственного решения.

Один из архитекторов также занимался разработкой пользовательского интерфейса системы. Он всегда имел четкое представление об интерфейсе, даже если мы не всегда соглашались по поводу того, в какой степени его представление можно было реализовать на практике. Он хотел, чтобы интерфейс был дружественным и наглядным. В интерфейсе не должны были использоваться меню – пользователи взаимодействуют с изображениями напрямую. Большие, яркие кнопки наглядно представляют все варианты. Короче говоря, рабочая станция должна была иметь интерфейс в стиле «киоск».

Оставалось принять решение относительно того, какая технология должна использоваться для вывода информации.

Один из участников нашей команды провел анализ доступных UI-технологий Java (как массовых, так и периферийных). Мы надеялись найти хорошую декларативную UI-инфраструктуру, нечто такое, что поможет нам избежать бесконечной возни с настройками Swing. Результат поразил всех.

В 2005 году, спустя 10 лет после появления Java, существовало две основные категории решений: «кошмар XML» и «спагетти GUI-строения». В решениях на базе XML компоненты Swing более или менее напрямую отображались на сущности и атрибуты XML. Нам это не подходило. Изменения в GUI требовали публикации новой версии кода независимо от того, реализовывались изменения в чистом коде Java или в файлах XML. Зачем держать в голове два языка – Java и схемы XML – вместо одного (Java)? Кроме того, с XML неудобно работать в программном коде.

Весь энтузиазм по поводу построителей GUI у нас пропал задолго до этого. Никто не хотел, чтобы бизнес-логика вплеталась в обработчики событий, встроенные в JPanel.

Мы с неохотой остановились на «чистом» Swing GUI, но с некоторыми дополнительными правилами. После нескольких обедов в местном ресторанчике Applebee был выработан новаторский механизм, который позволял нам использовать Swing, не увязая в его мелочах.

Пользовательский интерфейс и его модель

Типичная многослойная архитектура делится на уровни «представления», «предметной области» и «долгосрочного хранения данных». На практике основной код размещается на уровне представления, уровень предметной области превращается в набор анемичных контейнеров данных, а уровень долгосрочного хранения вырождается в вызовы функций инфраструктуры.

В то же время часть важной информации дублируется между уровнями. Например, максимальная длина фамилии представляется в виде ширины поля базы данных (возможно – в виде правила проверки данных в предметной области) и как параметр свойства `JTextField` в пользовательском интерфейсе.

В то же время в представление должна внедряться логика типа «если *этот* флажок установлен, снять блокировку с *этих* четырех текстовых полей». На первый взгляд кажется, что это утверждение относится к пользовательскому интерфейсу, но в действительности в нем отражена часть бизнес-логики: если посетитель является членом Портретного клуба, то приложение должно сохранить его личный номер и срок пребывания в клубе.

Таким образом, в типичной трехуровневой архитектуре часть информации распространяется между уровнями, а другая важная информация прячется в логике управления GUI.

В конечном итоге мы решили эту проблему инвертированием обычных отношений GUI с уровнем предметной области. Мы возложили на предметную область ответственность за разделение экранного представления и логических манипуляций со значениями и свойствами.

Формы

В нашей модели объект формы представляет атрибуты одного или нескольких объектов предметной области, оформленные в виде типизованных свойств. Форма управляет жизненным циклом объектов предметной области, а также обращениями к фасадам для поддержки транзакций и долгосрочного сохранения данных. Каждая форма представляет полный экран со множеством взаимодействующих объектов, хотя в некоторых ограниченных частных случаях возможно применение подформ.

Однако трюк заключается в том, что форма *абсолютно* невизуальна. Она не имеет дела с виджетами – только с объектами, свойствами и взаимодействиями между этими свойствами. Пользовательский интерфейс может связать логическое свойство с любой разновидностью представления: флажком, кнопкой-выключателем, текстовым полем или переключателем. Форме это совершенно безразлично. Она знает лишь о существовании свойства, способного принимать значения «истина/ложь».

Форма никогда не обращается с вызовами к своему объекту экрана. Большинству из них неизвестен даже конкретный класс своего экрана. Все взаимодействия между формами и экранами осуществляются через свойства и привязки.

Свойства

В отличие от типичных приложений на базе форм, свойства `Form` не относятся к примитивным типам Java или базовым типам вроде `java.lang.Integer`. Вместо этого `Property` хранит значение вместе с описывающими его метаданными. Объект `Property` способен ответить, содержит ли свойство одно значение или несколько, может ли оно принимать значение `null`, доступно ли оно в настоящий момент и т. д. Кроме того, он позволяет слушателям регистрироваться для получения событий.

Комбинация объектов `Form` и их объектов `Property` формирует четкую модель пользовательского интерфейса, никак не связанную с конкретными виджетами. Мы назвали этот уровень «UI-моделью» (рис. 4.4).

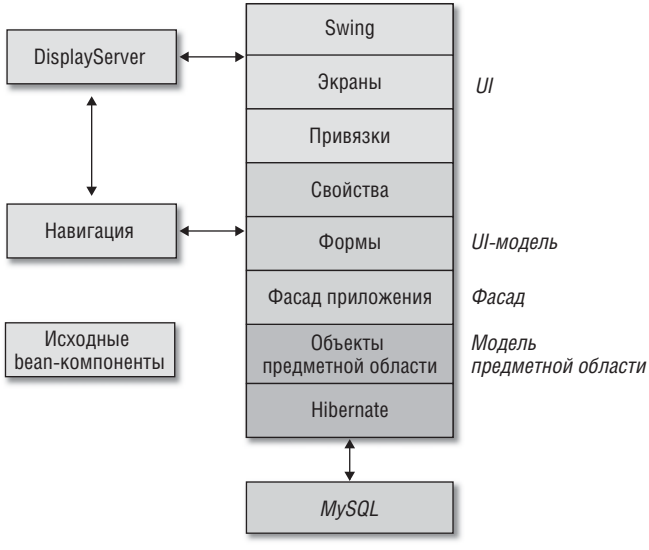


***Рис. 4.4.** Иерархическая архитектура*

Каждый подкласс `Property` работает с определенным типом значения. В конкретные подклассы включаются собственные методы для обращения к этим значениям. Например, `StringProperty` содержит методы `getStringValue()` и `setStringValue(String)`. Значения свойств всегда относятся к объектным типам, а не к примитивным типам Java, потому что примитивы не могут принимать значение `null`.

Может показаться, что классы свойств способны размножаться до бесконечности. Несомненно, так бы оно и было, если бы мы создавали класс свойства для каждого класса объекта предметной области. В большинстве случаев вместо прямого обращения к объекту предметной области `Form` предоставляет несколько свойств, представляющих разные аспекты объекта предметной области. Например, форма посетителя предоставляет объекты `StringProperty` для имени и фамилии посетителя, адреса, города и почтового индекса, а также объект `DateProperty` для хранения срока пребывания в клубе.

К некоторым объектам предметной области обращаться подобным способом неудобно. Соединение ползунка, управляющего растяжением изображения по заданной геометрической области, потребует более полудюжины свойств. Заставлять объект `Form` жонглировать таким количеством свойств ради простого перетаскивания ползунка было бы

сомнительно. Добавить новый тип свойства? Смахивает на неконтролируемое размножение типов.

Вместо этого мы пошли на компромисс и ввели объектное свойство для хранения произвольных объектов Java. Этому предшествовала оживленная дискуссия, с выражениями типа «скользкий путь» и «свалка объектов». К счастью, мы вовремя подавили свой порыв – вероятно, это была одна из вредных сторон языков с проверкой типов.

Для обработки действий было создано «командное свойство», инкапсулирующее объекты команд, а также определяющее их доступность. Таким образом, мы можем связать объект командного свойства с кнопкой GUI, используя изменения в состоянии свойства для установления или снятия блокировки кнопки.

UI-модель позволяет ограничить Swing уровнем UI. Кроме того, она обладает огромными преимуществами в области модульного тестирования. Наши модульные тесты могут управлять UI-моделью через свойства и делать предположения относительно изменений свойств, вызванных этими действиями.

Таким образом, сами формы не визуальны, но они предоставляют в распоряжение пользователя именованные свойства с сильной типизацией. Эти свойства где-то должны связываться с визуальными элементами. Этой задачей занимается уровень привязок.

Привязки

Если свойства соответствуют типам своих значений, привязки относятся к отдельным компонентам Swing. Объекты экранов создают свои компоненты, а затем регистрируют привязки для своих свойств к свойствам нижележащих объектов `Form`. Объект экрана не знает конкретный тип формы, с которой работает, – равно как и форма не знает конкретный тип экрана, с которым она связана.

Большинство привязок обновляет свои свойства при каждом изменении GUI. Текстовые поля, например, обновляются при каждом нажатии клавиши. Мы используем возможность оперативной проверки для предоставления постоянной, информативной обратной связи – вместо того, чтобы подождать, пока пользователь введет кучу неправильных данных и выдать гневное диалоговое окно.

Привязки также обеспечивают преобразование типа объекта свойства в разумное визуальное представление для соответствующих виджетов. Таким образом, привязка текстового поля знает, как преобразовывать целые числа, логические величины и даты в текст (и обратно). Впрочем, не каждая привязка способна обработать любой тип значения.

Например, не существует осмысленного преобразования графического свойства в текстовое поле. Мы позаботились о том, чтобы любые несоответствия перехватывались при запуске приложения.

Первый интересный момент обнаружился после того, как построили начальную версию инфраструктуры привязки свойств. Сначала мы опробовали ее на форме регистрации посетителей. Процесс регистрации весьма прямолинеен – пара текстовых полей, один флажок и несколько кнопок. Второй экран – альбомный – выглядит более эффектно и интерактивно. На нем размещается несколько виджетов: два эскиза, большой редактор изображения, ползунок и несколько кнопок. Даже здесь форма принимает решения относительно выделения, видимости и доступности виджетов исключительно на основании своих свойств. Таким образом, альбомная форма знает, что выделение эскизов влияет на центральный графический редактор, но объекту экрана об этом не известно. «Неведение» экранов помогает устранить ошибки синхронизации GUI и значительно расширяет возможности модульного тестирования.

Достаточно ли одной привязки?

На одних экранах эскизы допускают множественное выделение; на других – только единичное. Что еще хуже, некоторые действия возможны только при выделении ровно одного эскиза. Какой компонент должен решать, какую модель выделения следует применять и где должна сниматься блокировка с других команд на основании текущего выделения? Несомненно, эта логика относится к пользовательскому интерфейсу, поэтому она принадлежит уровню UI-модели. Иначе говоря, она принадлежит форме. UI-модель не должна импортировать классы Swing. Так как же форма выразит свои намерения относительно моделей выделения, не увязнув в коде Swing?

Мы решили, что у нас нет причин ограничивать компонент GUI всего одной привязкой. Иначе говоря, мы могли создать привязки, относящиеся к определенному аспекту компонента, и эти привязки могли связываться с разными свойствами форм.

Например, мы часто создавали отдельные привязки для представления содержимого виджета и его состояния выделения. Привязки выделения позволяли настроить виджет в режиме одиночного и множественного выделения в зависимости от привязанного свойства.

Хотя архитектуру привязки свойств приходится объяснять довольно долго, я все равно считаю ее одной из самых элегантных частей Creation Center. По своей природе Creation Center является визуальным приложением с расширенными пользовательскими взаимодействиями. Приложение предназначено для создания и обработки фотографий, поэтому оно просто не может напоминать стандартное бизнес-приложение с набором серых форм! Однако из небольшого набора простых объектов, каждый из которых определялся одной линией поведения, мы построили весьма динамичный интерфейс.

В конечном итоге клиентское приложение поддерживало перетаскивание мышью, выделение участков изображения, изменение размеров «на ходу», таблицы и активизацию двойным щелчком. И для этого нам ни разу не пришлось выйти за рамки архитектуры привязки свойств!

Фасад приложения

В процессе построения модели предметной области часто допускается типичная ошибка. Уровень представления – или, в нашем случае, UI-модель – часто делается слишком близкой к модели предметной области. Если представление переходит по отношениям в предметной области, это сильно затрудняет изменение модели предметной области. Мы, как и любая группа с динамической методологией разработки, должны были сохранять максимальную гибкость. Нам никак нельзя было принимать архитектурные решения, которые привели бы к уменьшению гибкости с течением времени.

Проблема решалась при помощи паттерна «Фасад приложения», описанного Мартином Фаулером (см. раздел «Библиография» в конце главы). Фасад приложения выводит на уровень представления ограниченную часть модели предметной области. Вместо того чтобы переходить по графам объектов предметной области, уровень представления обращается к фасаду приложения с запросами на содействие в перемещении, управлении жизненным циклом, активации и т. д.

Каждая форма определяла соответствующий фасадный интерфейс. В соответствии с каноном, который гласит, что интерфейсы должны определяться потребителями (а не поставщиками), мы поместили фасадный интерфейс в пакет формы. Форма обращается к фасаду с запросом на поиск объектов предметной области и их долгосрочное сохранение. В сущности, фасады управляли всеми транзакциями базы данных, так что формы вообще не знали о транзакционных границах.

Интерфейсы на границе между формами и фасадами также стали идеальным местом для изоляции объектов для модульного тестирования.

Чтобы протестировать некоторую форму, модульный тест создает фиктивный объект, который реализует интерфейс фасада. Тест приказывает фиктивному объекту передавать данные форме с некоторыми ожидаемыми результатами, включая условия ошибок, которые было бы очень трудно воспроизвести с реальным фасадом. Пожалуй, мы все рассматривали фиктивные объекты как двусторонний компромисс: хотя они обеспечивали возможность модульного тестирования, такая тесная привязка тестов к реализации форм все же вызывала беспокойство. Например, для фиктивных объектов приходилось задавать точную последовательность вызовов методов и точные значения параметров (более современные инфраструктуры обладают большей гибкостью). В результате с изменением внутренней структуры форм тесты переставали работать, хотя внешнее поведение при этом оставалось неизменным. В определенной степени это та цена, которую приходится платить за использование фиктивных объектов.

Во всех приложениях Creation Center (как студийных, так и в приложениях центра печати) использовался один и тот же стек уровней. Лишив GUI руководящей роли, мы избавились от бесконечной возни с настройками Swing. Инверсия управления также определила единую структуру, которой могло следовать каждое приложение. Хотя архитектура выходила за рамки обычного «трехслойного пирога», наш стек обеспечивал достаточно эффективное разделение сфер влияния: использование Swing ограничивалось областью UI, взаимодействие с предметной областью – формами, а долгосрочное хранение данных – фасадами.

Взаимозаменяемые рабочие станции

Завершив сеанс съемки, фотограф садится за любую свободную рабочую станцию. В зависимости от того, насколько в данный момент занята студия, он вместе с клиентом обычно завершает оформление заказа. Однако посетители часто возвращаются позднее, иногда даже в другой день. Было бы смешно навечно связывать посетителя с одной рабочей станцией – это не только неприемлемо в контексте планирования, но и рискованно. Рабочие станции ломаются!

Итак, все рабочие станции в студии должны быть взаимозаменяемыми, однако взаимозаменяемость создает некоторые проблемы. Объем графики одного сеанса может достигать гигабайта.

Сначала мы рассматривали возможность объединения рабочих станций в одноранговую сеть с распределенной репликацией. В конечном итоге была выбрана более традиционная модель «клиент/сервер», показанная на рис. 4.5.

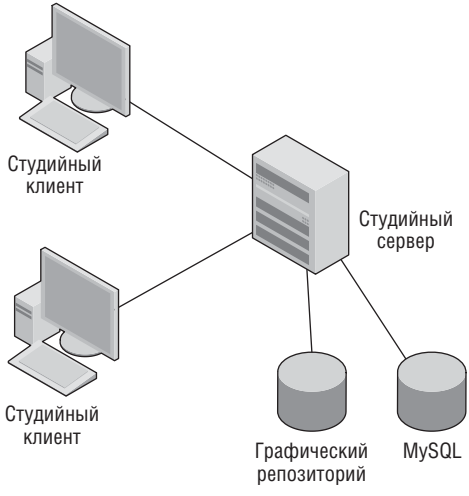


Рис. 4.5. *Структура студийной сети*

Сервер оснащается дисками большего объема, чем рабочие станции; диски объединяются в массивы RAID для защиты данных. На сервере работает база данных MySQL, используемая для хранения структурированных данных о посетителях, сеансах и заказах. Впрочем, большая часть дискового пространства выделяется для хранения фотографий.

Так как студии находятся на больших расстояниях, а техническая квалификация работников не гарантирована, мы понимали, что все подробности реализации должны оставаться невидимыми. Работники студий не должны работать с файловыми системами, исследовать причины сбоев или перезапускать задания. И, разумеется, нельзя требовать, чтобы они входили на сервер баз данных! В худшем случае, если сетевой кабель вышел из разъема, после его повторного подключения все должно снова заработать, а система должна автоматически восстановиться после сбоя.

Помня обо всем сказанном, мы взялись за планирование архитектуры системы и приложений.

Графический репозиторий

Чтобы рабочие станции стали действительно взаимозаменяемыми, было абсолютно необходимо обеспечить автоматическую пересылку изображений – как с рабочей станции, где они были загружены фотографом, на сервер, так и с сервера на другую рабочую станцию.

Как студийный клиент, так и студийный сервер используют центральный компонент, называемый графическим репозиторием. Он обеспечивает все аспекты хранения, загрузки и записи графических изображений вместе с их метаданными. На стороне клиента был построен локальный кэширующий посредник с отложенной записью. При получении запроса клиентский репозиторий либо возвращает изображение напрямую из локального кэша, либо загружает файл в локальный кэш, а затем возвращает его. В обоих случаях все происходит прозрачно для вызывающей стороны.

При добавлении изображений на рабочей станции клиентский репозиторий передает их на сервер. Мы используем пул программных потоков для выполнения передачи в фоновом режиме, чтобы пользователю не приходилось дожидаться ее завершения.

И клиентский, и серверный репозиторий интенсивно используют многопоточность. Мы создали систему блокировок, называемых «резервированием», – это упрощенная разновидность коллаборативной (совместной) блокировки. Чтобы поместить изображение в репозиторий, клиент должен сначала запросить и получить «резерв записи». Так мы можем быть уверены в том, что ни один поток не будет читать данные из графического файла при выдаче резервирования. Для чтения данных, естественно, придется получить «резерв чтения».

Хотя мы не стали реализовывать ни распределенные транзакции, ни двухфазное закрепление, на практике между предоставлением резерва записи клиентским репозиторием и соответствующего резерва записи серверной стороной существует минимальное окно. При получении второго резерва можно быть уверенным в том, что файлы не будут повреждены.

На практике даже конфликты блокировок крайне маловероятны. Для их возникновения необходимо, чтобы два фотографа на двух разных рабочих станциях обратились к сеансу одного посетителя. Тем не менее в каждой студии установлено несколько рабочих станций, а на каждой станции работает несколько потоков, так что осторожность не повредит.

Пересылка графики в NIO

Разумеется, остается проблема с пересылкой графики с клиента на сервер. Одним из вариантов, который мы рассмотрели и отвергли на ранней стадии, был механизм CIFS – сетевых дисков Windows. Основные опасения были связаны с отказоустойчивостью, но скорость передачи данных нас тоже беспокоила. Компьютеры должны были передавать огромные объемы данных, пока фотографы и посетители сидят рядом в ожидании.

В нашей матрице готовых решений ни один вариант не обеспечивал нужной комбинации скорости, параллелизма, отказоустойчивости и сокрытия информации. Неохотно, но мы все же решили создать собственный протокол передачи файлов, что привело нас к одной из самых сложных областей Creation Center. Передача графики стала суровым испытанием, но в конечном итоге нам удалось создать одну из самых надежных частей системы в целом.

У меня уже был некоторый опыт использования технологии Java NIO, и я знал, что мы можем использовать ее для создания быстрого механизма передачи графики. Построение пересылки данных NIO само по себе не было слишком сложным – мы воспользовались стандартным шаблоном «ведущий/ведомый» для обеспечения параллелизма, оставляя селекторные операции NIO в одном программном потоке.

Реализовать протокол было не так уж трудно, но нам приходилось учитывать множество нюансов:

- Сокет может быть закрыт на любом из концов, особенно в случае сбоя на стороне клиента. В коде примеров эта ситуация не обрабатывалась должным образом.
- Во время обработки события ввода/вывода `SelectionKey` продолжает сигнализировать о готовности. Это может привести к тому, что один обработчик будет вызван из нескольких потоков (если вы не позаботитесь о сбросе операции в *группе интересов* ключа).
- Ведущий должен выполнить все изменения в группе интересов `SelectionKey`, чтобы не возникла ситуация гонки с `Selector`. Соответственно нам пришлось построить очередь ожидающих изменений `SelectionKey`, которые должны быть выполнены ведущим потоком перед вызовом `select`.

Решение всех мелких технических проблем создало больше дополнительных логических привязок между объектами, чем я ожидал. Если бы мы создавали инфраструктуру, то надо всей областью пришлось бы основательно поработать. Но так как разрабатывалось приложение, мы посчитали, что совокупность взаимодействующих объектов можно рассматривать как единое целое.

Особенно интересный эффект проявился, когда мы установили перехватчик пакетов – нам хотелось узнать, насколько эффективно пересылаются данные по сети. Оказалось, что не очень. Когда реактор получал из сокета информацию о наличии данных, он читал один полный буфер, после чего возвращал управление. Оказалось, что студийная сеть работала достаточно быстро, чтобы заполнить окно TCP на сервере до того момента, как следующий поток получит доступ к обработчику, поэтому практически каждая передача данных простаивала

примерно половину времени. Мы включили в реактор цикл, который продолжал чтение до опустошения буфера. В результате время передачи данных уменьшилось почти наполовину, а непроизводительные затраты при переключении потоков и диспетчеризации сократились. Мне это показалось особенно интересным, потому что эффект наблюдается только в быстрых сетях с низкой задержкой и только при малом общем количестве клиентов. При более высокой задержке или большем количестве клиентов включение цикла создавало риск того, что некоторые клиенты будут простаивать в течение долгого времени. Еще раз подчеркну, что это было компромиссное решение, которое было оправдано в нашем конкретном контексте.

Я понимал, что создать быстрое, но хрупкое решение несложно. Настоящие проблемы возникнут с тем, чтобы сделать его надежным, особенно если вся сеть находится в студии за сотни миль от вас. И вы не можете использовать удаленный вход для диагностики проблем или устранения последствий сбоев. Студия с маленькими детьми, рассеянными родителями и серверами, стоящими на уровне детских глаз, – типичный случай неблагоприятной среды! Передать биты по кабелю недостаточно; нам нужна атомарность передачи файлов с гарантированной доставкой.

Первым уровнем защиты был сам протокол. При отправке файла с клиента на сервер в первый пакет запроса включалась контрольная сумма файла по алгоритму MD5. После отправки последнего пакета клиент ожидал ответа от сервера. Сервер отвечает одним из нескольких кодов: `OK`, `TIMEOUT`, `FAILED_CHECKSUM` или `UNKNOWN_ERROR`. При любом ответе, отличном от `OK`, клиент отправляет весь файл заново – мы назвали это «быстрым повтором». Клиент делает три быстрых повтора до того, как попытка передачи будет признана неудачной.

Проблемы с передачей файлов делятся на две разновидности. К первой относятся временные проблемы, которые решаются сами собой, – например, сетевые ошибки. Проблемы второго типа требуют человеческого вмешательства. Таким образом, проблема либо исчезает через несколько миллисекунд, либо на ее исправление требуется от нескольких минут до нескольких часов. Нет смысла повторять попытки передачи файлов снова и снова. Если файл не удалось передать с нескольких первых попыток, скорее всего, передача не будет работать в течение некоторого времени.

Следовательно, после исчерпания всех быстрых повторов клиент помещает задание на передачу файла в очередь. Фоновая задача активизируется каждые 20 минут и проверяет наличие незавершенных заданий передачи файлов. Она повторяет каждое задание еще раз, и в слу-

чае повторной неудачи возвращает его в очередь. Благодаря поддержке планирования заданий в Spring подобные «медленные повторы» реализуются почти тривиально.

Модульное тестирование и экспертная оценка кода

При разработке файлового сервера NIO я в очередной раз понял, насколько полезна экспертная оценка кода в больших группах, даже в проектах с гибкой методологией и полным делением на пары.

Мы с напарником большую часть итерации проработали над многопоточностью, блокировками и механизмами NIO. Мы проводили модульное тестирование всего, что могли, но на уровне многопоточности и низкоуровневого сокетного ввода/вывода трудно быть полностью уверенным в своем коде. Поэтому мы сделали лучшее, что могли: привлекли дополнительных рецензентов. Впрочем, ситуация была особой – мы компенсировали невозможность написания достаточно полных модульных тестов.

В общем случае две пары глаз, постоянно наблюдающие за кодом, предоставляют все преимущества экспертной оценки. Добавьте сюда автоматическое форматирование и проверку стиля – и затраты на проведение экспертной оценки кода начинают перевешивать ее преимущества. А если можно получить преимущества без затрат, то зачем возиться с ее проведением?

В нашей лаборатории был установлен проектор, подключенный к двум компьютерам через двухпозиционный переключатель. Каждый раз, когда нам требовалось описать использованный в коде прием или шаблон проектирования, мы тратили несколько послеобеденных минут на запуск проектора и просмотр кода. Это было особенно полезно на ранних стадиях, когда архитектура и композиция системы еще не зафиксировались, мы еще только разбирались в том, как работать с Spring и Hibernate. Кроме того, это помогло разобраться во многих тонкостях работы с Eclipse.

Проектор пригодился и для демонстрации промежуточных версий. Мы могли рассадить всех заинтересованных лиц в комнате, не заставляя их толпиться у одного монитора.

Я уже не говорю, насколько весело было проецировать клипы с YouTube на стену.

Сочетание быстрых и медленных повторов позволяет отделить операции сопровождения и поддержки на сервере от клиентских компьютеров. Обновление или замена не требует «холодной перезагрузки» всей студии.

Быстрота и надежность

Нам пришлось основательно потрудиться над локальными и удаленными репозиториями, а также сопутствующими механизмами передачи файлов. Но когда все было сделано, система отправляла изображения на сервер быстрее, чем они читались бы с карты памяти. Загрузка графики на другой компьютер происходила настолько быстро, что пользователи вообще ничего не замечали. Клиент загружал все эскизы альбома во время перехода от одного экрана к другому. Загрузка полноразмерных изображений могла осуществляться во время щелчка кнопкой мыши. С такой скоростью нам не было нужды выводить раздражающие диалоговые окна «Идет загрузка...»

Миграция баз данных

Представьте, что вы управляете 600 удаленными серверами в четырех часовых поясах. При этом любой сервер вполне может находиться на каком-нибудь необитаемом острове – собственно, с цифровой точки зрения они там и находятся. Если администратору баз данных потребуется внести изменения вручную, ему придется посетить сотню мест.

В таких обстоятельствах возможны разные варианты. Можно построить идеальную архитектуру базы данных непосредственно перед выпуском, чтобы никогда не изменять ее в будущем. Кто-то, может быть, еще считает, что это возможно, но в моей группе таких людей не было. Мы ожидали изменения на всех уровнях, включая базу данных, – и даже рассчитывали на них.

Другой вариант основан на рассылке сопроводительной документации к выпуску. При выполнении установки директор студии всегда обращался по телефону в службу поддержки за устными инструкциями. Теоретически мы могли разместить сценарии SQL в документах на компакт-диске, чтобы директор мог напечатать или скопировать их. Однако от одной мысли о том, как я буду диктовать: «А теперь наберите `mysqladmin -u root -p...`», меня бросает в холодный пот.

Вместо этого мы решили автоматизировать обновления базы данных. В Ruby on Rails они называются «миграциями баз данных», но в 2005 году эта технология еще не получила широкого распространения.

Обновления как объекты

Студийный сервер определяет bean-компонент, называемый *обновителем базы данных*. Он содержит список объектов обновления, каждый из которых представляет атомарное изменение базы данных. Каждое обновление базы данных знает свою версию и умеет применять себя к базе данных.

В процессе запуска обновитель базы данных проверяет текущую версию базы по таблице. Если таблица не обнаружена, он считает, что обновлений не существует или они уже были успешно применены. Соответственно самое первое обновление инициализирует таблицу версий и заносит в нее одну запись с номером версии и полем блокировки. Чтобы исключить одновременные обновления, обновитель базы данных сначала обновляет эту запись с установлением блокировки. Если установить блокировку не удается, то обновитель считает, что другой компьютер в сети уже занимается обновлением.

Мы использовали механизм миграции для применения как простых, так и относительно сложных изменений. Например, в одном из простых изменений мы добавили индексы для пары полей, влияющих на производительность. С другой стороны, нам пришлось изрядно поволноваться при преобразовании всех таблиц MyISAM к типу InnoDB (MyISAM, принятый по умолчанию тип таблиц в MySQL, не поддерживает транзакций и целостности ссылок, поддерживаемых типом InnoDB; если бы мы знали об этом до выпуска первой версии, то сразу воспользовались бы типом InnoDB). Так как базы данных при развертывании уже были заполнены реальными данными, нам пришлось использовать серию команд ALTER TABLE. Все прекрасно сработало.

После выпуска нескольких начальных версий было опубликовано около 10 обновлений. Все они прошли успешно.

Регулярная проверка

При каждой сборке локальная база данных сбрасывалась до нулевой версии, и к ней применялась обновления. Это означает, что механизм обновления ежедневно тестировался десятки раз.

Каждое обновление базы данных также подвергалось модульному тестированию. Каждый тестовый сценарий проверяет определенные допущения относительно состояния базы данных до обновления. Он применяет обновление, а затем проверяет допущения относительно нового состояния.

Тем не менее все эти тесты работали с «добропорядочными» данными. Однако в условиях реальной эксплуатации часто происходят странные

вещи, а с реальными данными проблем всегда больше, чем с любым набором тестовых данных. Наши обновления создают таблицы, добавляют индексы, заполняют таблицы записями и создают новые столбцы. Если структура данных окажется непредвиденной, некоторые из этих изменений могут привести к серьезным нарушениям. Мы беспокоились о риске обновлений и искали возможность сделать этот процесс более надежным.

Меры безопасности

Допустим, с одним из обновлений возникли какие-то проблемы. Студия может быть закрыта до тех пор, пока техническая служба не найдет способ восстановить базу данных, а если сбой будет действительно серьезным, база данных может быть повреждена или ее содержимое будет обновлено частично. В этом случае студия не сможет даже вернуться к предыдущей версии приложения. Для предотвращения подобных катастроф обновитель создает резервную копию базы данных перед обновлением. Если создать резервную копию не удается, обновитель прерывает процесс обновления.

Если в ходе обновления произойдет ошибка, обновитель автоматически пытается перезагрузить базу данных из резервной копии. Если даже это сделать не удастся, в студии хотя бы остается копия, и техник из службы поддержки может помочь директору с ручным восстановлением.

В самом худшем случае в центре печати всегда хранится копия базы данных, сделанная не более суток назад. Часть дополнительного пространства DVD с ежедневными заказами использовалась для отправки полной копии базы. Для небольшой базы данных и большого объема свободного места это было естественно.

Практические результаты

Время, затраченное на автоматизацию обновлений, не пропало даром. Во-первых, ранние обновления улучшили производительность и надежность системы. От пользователей была незамедлительно получена положительная обратная связь. Во-вторых, техническая служба высоко оценила простоту развертывания новых версий. Предыдущие системы требовали, чтобы студии пересылали туда-сюда съемные жесткие диски, со всеми вытекающими проблемами из области логистики. Наконец, наличие механизма обновления позволило нам сосредоточиться на построении минимально возможной структуры базы данных. Мы не заглядывали в магический кристалл и не пытались планировать схему базы данных на будущее. Вместо этого схема базы проектировалась в той степени, насколько этого требовала текущая итерация.

Неизменность данных и коды GUID

В ходе работы с посетителями фотограф создавал коллажи, в которых несколько фотографий встраивались в общий дизайн. Варианты дизайнов создавались дизайнерской группой в руководстве компании. Одни варианты универсальны, другие применяются только в определенное время года. Различные новогодние открытки отлично продаются, особенно в течение нескольких недель перед Новым годом. Не удивительно, что после праздника спрос на них резко падает.

Дизайн включает фоновое изображение, а также описание областей для размещения фотографий, их количества и геометрической формы. Работник студии проявляет свои творческие способности, заполняя пустые области фотографиями и другими изображениями.

Работа с дизайнами и базовыми изображениями, которые в них подставляются, создает целый ряд интересных проблем. Например, что произойдет, если посетитель разместит заказ, а в студию будет передана новая версия того же дизайна? Или что делать, если работник встроил один вариант дизайна в другой (скажем, фотографию в сепиевой цветовой гамме в рамку), а затем изменил или удалил исходный дизайн?

На первый взгляд казалось, что нас неизбежно ждет кошмар со счетчиками ссылок и скрытыми ссылками. Любая схема, которую мы рассматривали, базировалась на паутине из связей объектов со всеми вытекающими последствиями: ссылки, ведущие «в никуда», отсутствующие изображения или неожиданные изменения. Наша группа верила в «принцип наименьшего удивления», поэтому решение со скрытыми ссылками, из-за которых изменения в одном продукте могут распространяться на другие продукты, считалось неприемлемым.

Когда наш «старший архитектор» предложил простое, понятное решение, наша группа почти моментально с ним согласилась. Решение было основано на двух правилах:

1. Ничто не должно изменяться после создания. Дизайны и композиции должны оставаться неизменными.
2. Оригиналы должны копироваться (вместо простого создания ссылки).

В совокупности эти правила означают, что выбранный дизайн копируется в рабочее пространство. Если работник добавляет созданную композицию в альбом, то включается самостоятельная, независимая копия дизайна. Аналогичным образом, при вложении видоизмененного изображения создается копия оригинала. С момента вложения две композиции, исходная и новая, становятся полностью независимыми друг от друга.

Копирование не сводится к фокусам со ссылками на объекты в памяти. Описание композиции в формате XML содержит полную копию дизайна встроенных композиций. Это описание хранится в базе данных студии, и оно же отправляется в центр печати на DVD. Когда директор студии записывает ежедневные заказы на диск, студийный сервер упаковывает все необходимое для создания результата: исходные изображения, фоны, альфа-маски и инструкции по объединению компонентов.

Наличие полного описания всей композиции (в том числе и самого дизайна) на DVD имеет несомненные преимущества для печати.

В предыдущих системах дизайны хранились в библиотеке, а заказы просто ссылались на них по идентификатору. Это означало, что идентификаторы дизайнов должны были координироваться между студиями и центром печати. Таким образом, дизайны приходилось «регистрировать» перед рассылкой по студиям. Иногда синхронизация идентификаторов нарушалась, фотографии печатались в неверном дизайне, и посетители получали не то, что рассчитывали получить. Когда художники обновляли дизайн, в очереди производства оставались DVD со старым вариантом дизайна. Иногда замена проходила нормально, иногда – нет.

В новой системе варианты дизайна не требуют регистрации. Данные, поступившие в формате XML, отправляются непосредственно в производство; это позволяет дизайнерам вносить изменения в любой момент и распространять их тогда, когда они считают нужным. Изменение дизайна не влияет на заказы в очереди на производство, потому что каждый заказ полностью самостоятелен. Новая версия дизайна доходит до студий и постепенно начинает появляться в потоке заказов.

Не копировались только файлы изображений. Они были слишком большими, поэтому каждому изображению (как являющемуся частью дизайна, так и снятому в студии) присваивался собственный код GUID. Как правило, присваивание объекту кода GUID является фактическим объявлением его неизменности. Когда студийный сервер будет готов к записи заказов на DVD, он перебирает заказы и извлекает из них коды GUID (с применением известного паттерна «Посетитель»). Каждое найденное изображение записывается на DVD вместе с фотографиями посетителя и фонами дизайна.

Система визуализации

Студийный клиент помогает работникам изменять внешний вид базовых изображений. Усовершенствования могут быть как простыми (применение сепии или черно-белой гаммы для придания выразительности портрету), так и сложными (многоуровневые структуры с альфа-нало-

жением фрагментов фона, текста и мягкого фокуса). Независимо от применяемых эффектов итоговое изображение, выводимое на печать, не строится на рабочих станциях. Принтеры, установленные в центре печати, поддерживают разные размеры и разрешения. Работники центра могут свободно менять принтеры или в любой момент передавать задания с одного принтера на другой. Студии просто не обладают достаточной информацией для построения изображений, готовых к печати.

Содержимое дисков, ежедневно отправляемых студиями в центр печати, загружается в управляющую систему PCS (Production Control System). PCS принимает все решения относительно того, когда строить изображения для заказа, когда и на каких принтерах их печатать. PCS разрабатывалась отдельной группой, находящейся в другом месте и в другом часовом поясе. В предыдущих проектах попытки слишком тесной интеграции с PCS приводили к серьезным трениям. Все стороны действовали с лучшими намерениями, но трудности общения замедляли работу обоих групп. Чтобы избежать этих трений, мы решили воспользоваться законом Конуэя (см. следующий раздел) и сформировать в программном продукте интерфейс по границе между рабочими группами.

Закон Конуэя

Закон Конуэя часто упоминают «задним числом», чтобы объяснить структурное деление продукта, которое на первый взгляд кажется произвольным. Закон утверждает фундаментальную истину, относящуюся к деятельности групп разработки: везде, где существует граница между группами, появляется структурная граница в программном продукте. Она появляется из-за необходимости согласования интерфейсов.

Мы решили, что формат и структура DVD должны находиться под полным контролем Creation Center, поэтому в нашу часть проекта была включена новая программа DvdLoader. Эта программа, работающая в центре печати, предназначена для чтения DVD и вызова различных сохраненных процедур PCS для добавления заказов, композиций и изображений. PCS рассматривает инструкции по формированию композиции как монолитную последовательность символов, и мы тщательно избегали любых решений, которые заставили бы PCS извлекать из этой строки код XML. Иногда такой подход приводил к дублированию информации, но это было приемлемой компенсацией за поддержание четкой границы.

Аналогичным образом был определен интерфейс, который позволял системе визуализации получать задания у PCS, притом что описание самого процесса построения в формате XML оставалось под контролем Creation Center. Мы разработали письменные спецификации этих интерфейсов, а затем воспользовались инфраструктурой FIT на нашем

Пошаговое формирование архитектуры

В сообществе гибких методологий постоянно встречается один вопрос: «Какая часть архитектуры должна создаваться заранее?» Некоторые теоретики на это отвечают: «Никакая. Безжалостно перерабатывайте свой код, и архитектура сформируется сама». Я никогда не принадлежал к этому лагерю.

Переработка улучшает архитектуру кода без изменения его функциональности. Но чтобы улучшить архитектуру, необходимо отличать хорошее от плохого. У нас уже имеется хороший каталог «запахов кода», которым мы можем руководствоваться, но аналогичные каталоги «запахов архитектуры» мне неизвестны. Кроме того, должна существовать возможность непрерывного внесения изменений даже через границы интерфейсов. Из-за этого я всегда считал, что фундаментальная архитектура системы должна быть сформирована до начала разработки.

Теперь, после проекта Creation Center, я уже не так уверен в этом ответе. Многие серьезные архитектурные блоки были добавлены на относительно поздней стадии проекта. Примеры:

- Поддержка Hibernate, добавлена после двух или трех итераций. До этого момента мы не использовали базы данных.
- Поддержка Spring, добавлена почти на трети пути к версии 1.0, быстро заняла центральное место в нашей архитектуре. Даже не представляю, как мы обходились без нее.
- Поддержка FIT, добавлена на половине пути к версии 1.0.
- Программа записи DVD, куплена и добавлена в конце стадии начальной разработки.
- Поддержка оконных интерфейсов, добавлена за две итерации до выпуска.

В каждом случае перед принятием решения мы тщательно исследовали все доступные варианты. Решение принималось в «последний ответственный момент» – когда затраты от непринятого решения перевешивали затраты на его реализацию. Возможно, если бы поддержка Spring присутствовала в проекте с самого начала, кое-что было бы сделано иначе, но ее добавление на более поздней стадии обошлось без вреда.

Во время ранних итераций наше внимание было направлено на то, каким мы хотели видеть наше приложение – а не на то, как оно должно строиться по канонам Spring.

сервере разработки для «уточнения» их смысла. По сути, мы использовали FIT как исполняемую спецификацию интерфейсов. Позднее эта функция сыграла важную роль, потому что даже люди, занимавшиеся согласованием интерфейсов, обнаруживали расхождения между тем, что они считали уже согласованным, и фактическими реализациями. Применение FIT позволило нам устранить несоответствия в ходе разработки, а не в ходе интеграционного тестирования – или, еще хуже, в ходе реальной эксплуатации.

Загрузка данных с DVD

Программа DvdLoader, работающая в центре печати, фактически представляет собой пакетный обработчик, который читает заказы с DVD и загружает их в PCS. Как и во всех остальных случаях, первоочередное внимание уделялось надежности работы.

DvdLoader читает весь заказ и проверяет наличие всех составляющих перед его передачей в PCS. В этом случае в базу данных не попадают неполные или поврежденные заказы.

Поскольку некоторые изображения могут присутствовать на многих DVD, загрузчик сначала проверяет, не было ли ранее загружено в базу изображение с таким же кодом GUID. Если такого изображения нет, загрузчик добавляет его в базу. Следовательно, при необходимости студия может прислать заказ повторно, даже если система PCS уже стерла сам заказ и используемую им графику. Кроме того, это означает, что фоновые изображения, используемые в дизайне, будут загружены в базу при первом поступлении заказа.

Таким образом, содержимое DVD полностью автономно и самостоятельно.

Конвейер визуализации

Что касается самой системы визуализации, мы воспользовались классической архитектурой с каналами/фильтрами. Метафора «конвейера» естественным образом подходит для построения изображений, а разделение сложной последовательности действий на более простые шаги упрощает модульное тестирование.

При извлечении задания из PCS система визуализации создает объект `RenderRequest`. Объект передается конвейеру визуализации, каждая стадия которого работает с самим запросом. Одна из завершающих стадий конвейера сохраняет построенное изображение в каталоге, путь которого задает PCS. К моменту выхода из конвейера запрос содержит только объект результата с признаком успешного завершения и (возможно) набором сообщений об ошибках.

На каждом этапе конвейера возможны сообщения о возникших проблемах, включая сообщение об ошибке в объект результата. Если на каком-то этапе возникнут ошибки, работа конвейера прерывается, а информация о проблеме передается PCS.

Быстрый отказ

В каждой системе возможны сбои; вопрос лишь в том, учитываются ли они при проектировании или происходят сами собой. Мы позаботились о том, чтобы интегрировать средства безопасности в свою архитектуру, и, прежде всего, в процесс печати. Наш продукт ни в коем случае не должен был нести ответственность за остановку производственного процесса.

Также был и другой аспект. Посетитель должен получить именно то, что ожидал получить! Иначе говоря, конечный продукт должен соответствовать его заказу. На первый взгляд это утверждение кажется очевидным, но на самом деле очень важно, чтобы печатное изображение полностью соответствовало его предварительной копии на экране, одобренной посетителем. Мы основательно потрудились над тем, чтобы при печати использовался тот же код визуализации, что и при предварительном просмотре в студии. Кроме того, при печати должны были использоваться те же шрифты и фоны.

При проектировании ядра визуализации мы следовали философии «Отказы выявляются быстро, отказы хорошо заметны». Получив задание от PCS, механизм визуализации проверяет все инструкции и убеждается в доступности всех ресурсов, необходимых для обработки задания. Если в задании присутствует текст, система визуализации немедленно загружает соответствующие шрифты. Если в задании используются фоновые изображения или альфа-маска, соответствующие изображения тоже загружаются немедленно. При отсутствии каких-либо ресурсов ядро визуализации немедленно оповещает PCS об ошибке и отменяет задание. Из 16 этапов конвейера визуализации первые 5 относятся исключительно к проверке.

По прошествии нескольких месяцев эксплуатации была выявлена ошибка, которая не обнаруживалась на ранних стадиях: конвейер визуализации не резервировал заранее дисковое пространство для итогового изображения. Когда все запасы дискового пространства PCS были исчерпаны, сбои заданий печати начали происходить на поздней стадии вместо ранней. За все предшествующее время не было ни одного исправления, связанного с ошибками при построении изображений.

Масштабирование

Каждое ядро визуализации работает независимо от других. PCS не ведет список существующих ядер визуализации; каждое ядро просто получает задания от PCS. Более того, ядра могут добавляться и удаляться по мере надобности. Поскольку каждое ядро начинает искать новое задание сразу же после выполнения предыдущего, в системе автоматически обеспечивается распределение нагрузки в соответствии с производительностью отдельных ядер. Более быстрые ядра визуализации поглощают задания с более высокой скоростью. Их разнородность не создает никаких проблем.

Единственным «узким местом» будет сама система PCS. Так как ядра визуализации вызывают сохраненные процедуры для получения заданий и обновления состояния, каждое ядро генерирует две транзакции каждые три-пять минут. PCS работает на довольно внушительном кластере хостов Microsoft SQL Server, так что в ближайшем будущем вряд ли стоит ждать проблем с пропускной способностью системы.

Реакция пользователей

Первая версия была установлена в двух близлежащих студиях, куда можно было доехать на машине для отладки. Реакция пользователей была незамедлительной и весьма положительной. По оценке одного директора студии, новая система оказалась настолько быстрее и удобнее в работе, что во время праздников студия сможет обслужить на 50% больше посетителей. Одного из работников студии даже спросили, где можно купить копию программы. Нам часто сообщали о том, как посетители сами берутся за мышь и вносят собственные усовершенствования. Естественно, посетитель с гораздо большей готовностью закажет тот продукт, который он создал сам.

В нашем процессе разработки были свои трудности, но все проблемы решались очень быстро. Благодаря гибкости, заложенной нами при проектировании загрузчика и подсистемы визуализации, центр печати смог справиться с нагрузкой от большего количества студий, чем предполагалось изначально, и с повышением эффективности работы.

Заключение

Я мог бы потратить много времени и места, подробно описывая каждый класс, каждое взаимодействие или архитектурное решение, – примерно так, как неопытный родитель описывает каждую выходку своего первенца. Вместо этого в данной главе в сжатом виде представ-

лены результаты годичных усилий, исследований, пролитого пота и крови. Глава показывает, как структура и динамика архитектуры Creation Center формировалась под влиянием движущих сил и контекста бизнеса. Благодаря хорошему разделению сфер влияния в архитектуре, а также управлению пошаговым проектированием и разработкой нам удалось достаточно хорошо сбалансировать эти силы.

Библиография

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt. 2007. «*Pattern-Oriented Software Architecture: A Pattern for Distributed Computing*», vol. 4. Hoboken, NJ: Wiley.

Fowler, Martin. 1996. «*Analysis Patterns: Reusable Object Models*». Boston, MA: Addison-Wesley.

Fowler, Martin. «Application facades»/ *http://martinfowler.com/apsupp/appfacades.pdf*.

Gamma, Erich, et al. 1994. «*Design Patterns: Elements of Reusable Object-Oriented Software*». Boston, MA: Addison-Wesley.[1]

Hunt, Andrew, and David Thomas. 1999. «*The Pragmatic Programmer*». Boston, MA: Addison-Wesley.[2]

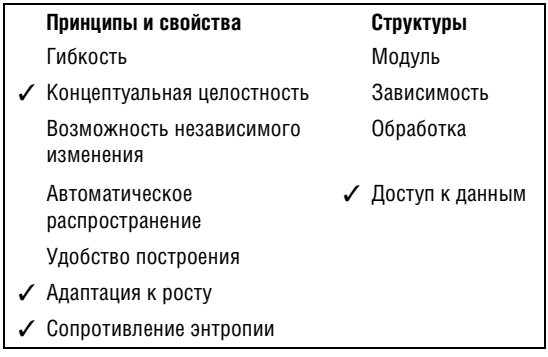
Lea, Doug. 2000. «*Concurrent Programming in Java*», Second Edition. Boston, MA: Addison-Wesley.

Martin, Robert C. 2002. «*Agile Software Development, Principles, Patterns, and Practices*». Upper Saddle River, NJ: Prentice-Hall.[3]

[1] Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001–2007.

[2] Э. Хант, Д. Томас «Программист-прагматик», Лори, 2009.

[3] Р. Мартин «Быстрая разработка программ. Принципы, примеры, практика», Вильямс, 2004.



5

Ресурсно-ориентированные архитектуры, жизнь в WWW

Брайан Слеттен

Архитектура – это обитаемая скульптура.

Константин Бранкузи

В этой главе будет показано, что информационно-ориентированным архитектурам корпоративного уровня присущи некоторые из положительных свойств, характерных для Сети: масштабируемость, гибкость, стратегии архитектурной миграции, управление доступом на основании информации и т. д. Заодно они позволяют бизнес-стороне принимать решения относительно капиталовложений и разработки программных продуктов на основании своих экономических потребностей (а не просто из-за неудачного выбора технологий).

Введение

Мы, представители IT-индустрии, вынуждены с величайшим стыдом признать один неприятный факт: для большинства организаций проще найти нужную информацию в Сети, чем в своих собственных системах.

Только подумайте: найти данные в глобальной информационной системе (пусть и при содействии третьих сторон) оказывается проще, чем сделать то же самое в среде, находящейся под вашим полным контролем. Парадокс возникает по многим причинам, но главная проблема заключается в том, что мы в своих представлениях склонны использовать неверные абстракции, переоценивая свои программы и предоставляемый сервис, но недооценивая роль данных. Этот принципиально неверный подход в значительной мере объясняет, почему экономические подразделения наших организаций вечно недовольны IT-отделами. Мы забываем, что компаниям нет никакого дела до наших программ – их интересуют только те функции и возможности, которые эти программы предоставляют. В действительности бизнесу нужны более простые средства для обработки собранных данных, принятия решений на их основе и повторного использования для поддержки клиентов.

Почему же управление информацией в организациях так отличается от управления информацией в Сети? К сожалению, проблема обусловлена не только выбором технологии, но и корпоративной политикой. Мы имеем дело с унаследованными системами, усложняющими применение современных идиом взаимодействия. Мы пытаемся использовать решения, разработанные фирмами, интересы которых не всегда соответствуют нашим. Мы хотим найти «серебряную пулю» для решения всех своих проблем (хотя доктор Брукс уже давно предостерегал нас от этого[1]). Даже если вы каким-то чудом окажетесь в организации с идеально сбалансированной технологической инфраструктурой, поставщики и потребители данных часто ведут между собой войны «на захват территории», препятствующие нормальному обмену информацией. Это одна из причин, по которым информационные системы в компаниях работают не так четко, как в Сети: у них нет стимула к обмену информацией, хотя необходимость в нем определенно существует. Из всего сказанного следует, что не все проблемы имеют техническую природу. В определенной степени веб-технологии помогают обойти и политические проблемы, потому что вам не всегда требуются специальные разрешения для публикации ссылок на информацию, доступную в других формах.

К счастью, мы можем обратиться к Сети и попытаться определить, что же делает ее такой замечательной средой для поиска информации. Применение этих концепций в организациях поможет решить упоминавшиеся проблемы, а также принесет другие преимущества того же уровня: экономичное управление данными, стратегии архитектурной

[1] *http://en.wikipedia.org/wiki/No_Silver_Bullet*

миграции, управление доступом в зависимости от информации и соблюдение нормативных ограничений. Успех WWW в значительной мере обусловлен тем обстоятельством, что эта среда расширила возможности обмена информацией, одновременно сделав информацию более доступной. Появились инструменты и протоколы, при помощи которых ведущие ученые мира обмениваются знаниями – а наши бабушки связываются со своими семьями, находят интересующие их материалы и сообщества. Это весьма серьезное достижение; стоит подробнее разобраться в том, какое сочетание идей привело к этому результату. Нам приходится жить с теми архитектурами, которые мы создаем, поэтому создаваемые архитектуры должны одновременно радовать и вдохновлять нас.

Традиционные веб-службы

Прежде чем приступать к изучению новой архитектуры информационных систем, мы в общих чертах посмотрим, как эти системы строились в последнее время и что можно было бы улучшить в их архитектуре. За последние 10 лет сформировалось господствующее представление о корпоративных архитектурах, построенных на концепции бизнес-служб, рассчитанных на повторное использование. Однако не стоит забывать о том, что веб-службы были задуманы как бизнес-стратегия, как способ определения функциональности, доступной из любого места, на любом языке, в асинхронном режиме. Мы хотели, чтобы обновление служб не отражалось на клиентах, которые их используют. К сожалению, обширный и постоянно изменяющийся арсенал технологий, связанных с этой целью, сбил с толку людей и не решил тех проблем, с которыми мы сталкиваемся в реальных архитектурах реальных организаций. В своем новом подходе мы постараемся расширить возможности и улучшить репутацию неудачных служебно-ориентированных архитектур, которые встречались нам в прошлом.

Наши базовые представления о веб-службах складываются из нескольких технологий: SOAP для обращения к службе, WSDL для описания контракта, UDDI для публикации и получения метаданных службы. Технология SOAP выросла из многих традиций, включая модель RPC (Remote Procedure Call) и модель асинхронного обмена сообщениями XML (doc/lit). Первая слишком хрупка, плохо масштабируется и не так уж хорошо проявила себя под предыдущими названиями DCOM, RMI и CORBA. Угловые скобки не создают и не решают проблем; просто системы строились на неверном уровне детализации и с преждевременной привязкой к контракту, который явно не должен был оставаться статическим. Вторая модель была более прогрессивной и обес-

печивала хорошую стратегию реализации, однако она не соответствует всей шумихе об операционной совместимости, сопровождавшей ее с самого начала. Она усложняет даже простые взаимодействия, потому что на ее процессы влияет стремление к решению более серьезных проблем.

Стиль doc/lit позволяет определить запрос в структурированном пакете, который может пересылаться, дополняться и обрабатываться слабо связанными участниками рабочего процесса. Сообщение, обрабатываемое посредниками и конечными точками в потенциально асинхронном стиле, обрастает элементами и атрибутами. Горизонтальная масштабируемость достигается введением еще большего количества обработчиков сообщений на одном уровне. Модель позволяет стандартизировать взаимодействия на партнерских и отраслевых границах с бизнес-процессами, не умещающимися в одном контексте. Она представляет лишенный контекста запрос, подходящий даже для очень сложных схем взаимодействий.

Когда жесткой декомпозиции служб и описаний (т. е. SOAP и WSDL) оказывалось недостаточно для наших потребностей во взаимодействиях, мы поднимались по стеку технологий и вводили новые уровни бизнес-обработки и управления. Появление многочисленных стандартов и инструментов усложнило и без того выходящую из под контроля ситуацию. При выходе за границы предметных областей и организаций возникают конфликты терминов, бизнес-правил, политик доступа... настоящая Вавилонская башня. Но даже если подчиниться этим правилам, у вас все равно не будет реальных стратегий миграции, а все разговоры об операционной совместимости по сути оказываются враньем. Клей Ширки (Clay Shirky) остроумно охарактеризовал операционную совместимость веб-служб фразой «черепахи до самого верха[1]».

Как правило, когда люди хотят использовать некую многоразовую функциональность независимо от языка и платформы, эти технологии оказываются слишком сложными и отягощенными подробностями реализации. Чтобы использовать такую функциональность, необходимо «говорить на языке SOAP». Возможно, это неплохой выбор реализации, однако в нашем мире слабосвязанных систем мы не всегда хотим сообщать клиентам такие подробности простых взаимодействий (а также требовать их соблюдения).

Общая концепция применения SOAP подразумевает отсутствие контекста у запросов и обеспечение транзакционной целостности в асинхронной среде. Однако в реалиях современных систем запросам снова

[1] *http://en.wikipedia.org/wiki/Turtles_all_the_way_down*

приходится возвращать контекст. Сначала запрос ассоциируется с личностью пользователя, потом с его регистрационными данными, затем сообщение снабжается электронной подписью и в нем шифруется конфиденциальная информация и т. д. «Простое» дело выдачи запросов SOAP отягощается стилем взаимодействий и нашими бизнес-потребностями. Если кто-то в организации хочет получить некую информацию, почему он просто не запросит ее? А когда на его запрос будет получен ответ, почему 10 (или 100, или 1000) людей, задающих тот же вопрос, должны создавать нагрузку на служебную подсистему своими повторяющимися запросами?

Все эти вопросы наглядно подчеркивают некоторые проблемы абстракций, присущие традиционному технологическому стеку веб-служб. Они, по крайней мере, частично объясняют прохладное отношение к веб-службам в IT-отделах по всему миру. Технологии веб-служб предназначены для практического разложения некоторого поведения на упорядоченные потоки операций служб, однако полная номенклатура потребностей организации не может быть выражена только в контексте служб. Мы теряем возможность идентификации и структурирования информации в контексте непосредственного вызова. Мы не можем просто запросить нужную информацию без понимания технологий, используемых для ее выборки. Привязка к запросам, подчиняющимся жесткому контракту и работающим на конкретном порте конкретной машины, не позволяет использовать преимущества слабой связанности и асинхронных паттернов, а также способности адаптации к изменяющимся представлениям данных. Без возможности однозначной идентификации данных, передаваемых между службами, мы не можем применять контроль доступа на уровне информации. Это усложняет и без того непростую задачу защиты доступа к конфиденциальной, ценной информации в современном мире, в котором сети проникают все глубже в повседневную жизнь.

SOAP и WSDL не создают проблем, но и не обеспечивают полноценных решений. Скорее всего, мы будем продолжать использовать SOAP в стиле doc/lit в ресурсно-ориентированных архитектурах, которые будут описаны в этой главе; просто не стоит относиться к ним как к единственно возможному решению. Также не стоит без крайней необходимости открыто объявлять о том, что эти технологии используются во внутренней реализации. Чтобы сделать следующий шаг, необходимо присмотреться к WWW и понять причины ее успеха как масштабируемой, гибкой, способной к развитию платформы совместного доступа к информации. Подробности реализации часто не интересуют потребителей информации.

WWW

В наших представлениях о Сети преобладает документно-центрическая модель. Мы думаем о WWW в контексте получения документов броузерами, потому что именно в этом заключается суть наших взаимодействий. Однако настоящее волшебство WWW проявляется в открытом связывании общедоступной информации и простоте создания «окон» для представления нижележащего контента. У Сети нет ни начала, ни конца. Если вы знаете, какая информация вам нужна, обычно вам удается ее получить. Появились технологии, которые помогают уточнить, что именно вы ищете, – либо через поисковые системы, либо через каталоги рекомендаций.

Склонность давать имена заложена в человеческую природу; мы используем имена, чтобы отличить «это» от «того». Свои первые коммуникационные операции мы совершаем в раннем детстве, когда указываем на интересующие нас предметы и требуем дать их нам. WWW во многих отношениях является применением этого «детского» подхода к нашей коллективной мудрости (и глупости). Человек – существо с неутолимой жаждой знаний, он просто решает, что его интересует, и требует дать ему «это». Никакой централизации не существует; ничто не мешает вам документировать историю своих странствий в Сети. Мы представляем себе WWW в виде совокупности однонаправленных ссылок между документами (рис. 5.1).

Однако связывание документов – лишь часть картины. Наше представление о Сети всегда включает идею связывания данных. Контент

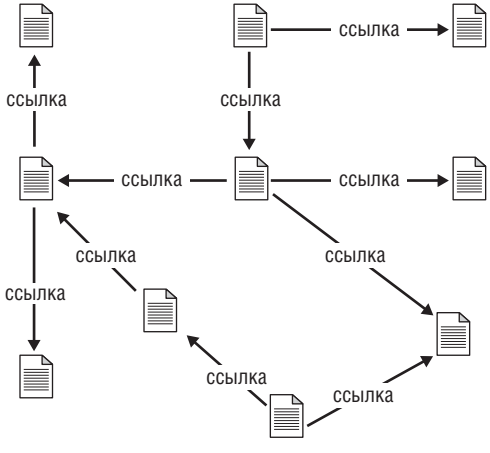


Рис. 5.1. *WWW в традиционном представлении*

может потребляться через визуальное представление, но также возможна его прямая обработка в нужных формах в других контекстах. Представьте средний уровень, запрашивающий информацию в формате документа XML, притом что уровень представления предпочитает объект JSON, для получения которого используется вызов AJAX. Одним именем обозначаются одни данные в разных формах. Обеспечив возможность такой адресации, вы сможете легко строить многоуровневые приложения с логически целостными представлениями данных, даже на разных уровнях детализации или с особыми требованиями к представлению. Приложения и среды, которые производят и потребляют данные в подобном «слабосвязанном» стиле, считаются «соответствующими парадигме Web». Мы постепенно движемся по направлению к Паутине данных, связывающей людей, документы, данные, сервис и концепции (рис. 5.2).

Основным видом взаимодействия в этой среде является логический запрос «клиент–сервер». Интересующая нас информация обозначается адресом. Идентификатор URL (Uniform Resource Locator) не только используется для разрешения ссылок в глобальном адресном пространстве, но и указывает, как должна быть устроена ссылка. Ни в од-

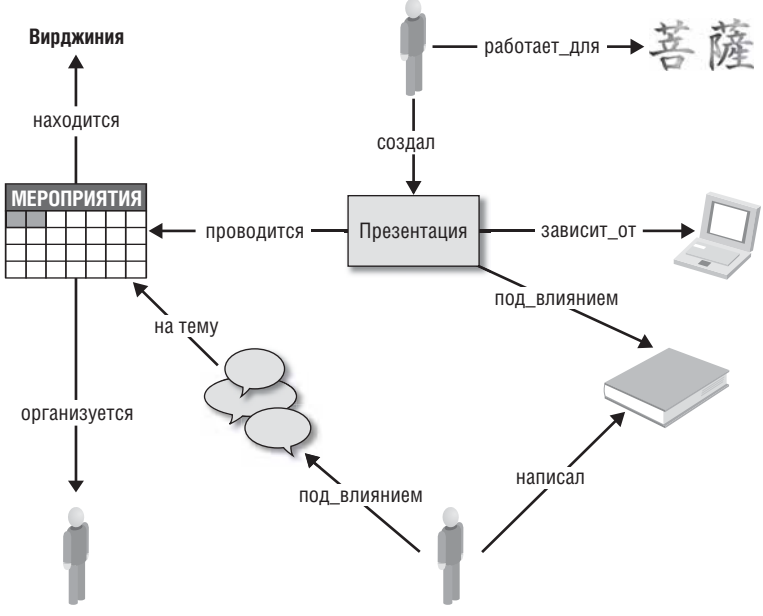


Рис. 5.2. *Паутина данных*

ной точке процесса для удовлетворения запроса не требуется понимание используемых технологий. Процесс остается простым и максимально гибким в отношении изменения реализации. Когда наши любимые сайты переходят со статических данных на динамические или меняют поставщиков серверных приложений, мы этого не замечаем. Хотя многие сайты не обеспечивают согласования контента, у нас, по крайней мере, сохраняется потенциальная возможность получения разных представлений одноименных сущностей. Позднее будет показано, как воспользоваться этим свойством для определения уровня детализации с целью управления доступом и соблюдения нормативных ограничений.

Используемые в Сети схемы присваивания имен позволяют идентифицировать документы, данные, сервисные функции и даже концепции. Традиционно пользователю было трудно отличить ссылку, скажем, на Авраама Линкольна от ссылки на посвященный ему документ. Так, сайт *http://someserver/abrahamlincoln* может быть и тем, и другим. Группа технической архитектуры W3C (TAG) выдвинула рекомендацию[1], согласно которой ресурсы, не имеющие сетевой адресации (т. е. сущности, не живущие непосредственно в Web, но представляющие интерес для запрашивающей стороны), могут обозначаться кодом ответа 303 вместо привычного кода 200. Этот ответ как бы говорит клиенту: «Да, ваш запрос вполне допустим и законен, но этот ресурс в Web не живет. Вы можете получить дополнительную информацию по следующему адресу...»

Веб-адреса начинаются со ссылки на протокол HTTP; за ней следует имя сервера, отвечающего на запрос. Далее идет иерархическая схема, представляющая некий путь в информационном пространстве. В сущности, это логическое имя, содержащее информацию о структуре данных. Множественные пути могут вести к одному ресурсу, но использоваться в разных ситуациях. Допустим, *http://server/order/open* возвращает список всех открытых заказов на определенный момент времени, а адрес *http://server/order/customer/112345* отражает все открытые заказы определенного клиента. Конечно, результаты, возвращаемые этими двумя логическими запросами, будут перекрываться. Если мы не знаем, что именно следует запрашивать, можно пойти по более общему пути. Если мы хотим запросить информацию о состоянии конкретного клиента, идем напрямую. Логические URL-ссылки либо передаются из другой части системы, либо генерируются на основании данных, вводимых клиентом в пользовательском интерфейсе.

---

[1] *http://lists.w3.org/Archives/Public/www-tag/2005Jun/0039*

Разделение сфер влияния проходит по ключевым абстракциям стиля взаимодействия. Обрабатываемые ресурсы отделяются от операций, при помощи которых мы манипулируем с этими ресурсами, и от форм, используемых для получения и отправки данных. Этот принцип продемонстрирован на рис. 5.3, позаимствованном из обсуждения в REST-Wiki[1]. В архитектурном стиле REST[2] (REpresentational State Transfer) выделяются ресурсы (существительные), операции (глаголы) и представление ответа. Ресурсом может быть любая сущность, которая может адресоваться (в том числе и концепции!). Основные операции – GET (выборка), POST/PUT (создание/обновление) и DELETE (удаление). Согласно установленным ограничениям операции GET не должны иметь побочных эффектов. Это свойство называется «идемпотентностью запроса». Семантика таких взаимодействий открывает возможности для кэширования. Операции POST обычно используются при отсутствии центрального органа, способного ответить на запрос (как, например, при отправке новостей в сообществе Usenet) или при отсутствии возможности адресации нужного ресурса. Невозможно идентифицировать заказ до того, как он будет создан, потому что за создание идентификаторов заказов отвечает серверное приложение. Соответственно эти запросы передаются операцией POST блоку функциональности (например, сервлету), который принимает запрос и генерирует идентификатор. Операции PUT используются для обновления и замены существующего состояния ресурса с заданным именем. Операция DELETE не находит особого применения в общедоступном сегменте Web (к счастью!), но играет важную роль для управления жизненным циклом ресурсов в ресурсно-ориентированных средах с внутренним управлением – она показывает, что некоторый ресурс более не представляет для нас интереса. Стиль REST базируется на разделении

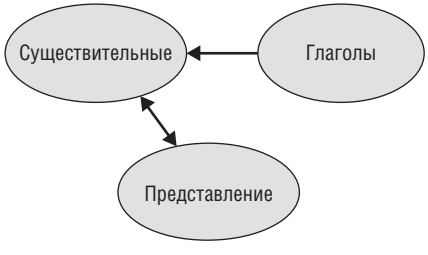


Рис. 5.3. *Логическое деление в архитектуре REST*

[1] *http://rest.blueoxen.net/cgi-bin/wiki.pl?RestTriangle*

[2] *http://en.wikipedia.org/wiki/REST*

логических имен интересующих нас ресурсов, средств для работы с ними и форматов их представления (рис. 5.3).

Такое деление кардинально отличается от контрактной природы обращений к службам SOAP, в которых структура запроса, активизируемое поведение и форма возвращаемого типа часто связываются с контрактом на уровне языка WSDL (Web Services Definition Language). Не стоит думать, что контракты плохи; они полезны до тех пор, пока мы не пытаемся выйти за их пределы. Одной из основных целей технологического стека веб-служб было сокращение логических привязок и введение асинхронной модели обработки, при которой обработчик сообщения может быть обновлен в соответствии с новой бизнес-логикой, не оказывая влияния на стороне клиента. Авторы подхода, основанного на WSDL-привязках, заметили эту цель – и сделали нечто прямо противоположное. В общем случае невозможно изменить привязку на порте, чтобы это изменение не отразилось на работе клиента (а ведь именно этого мы пытались избежать!)

Ресурсно-ориентированный подход позволяет нам устанавливать контракты тогда и в тех местах, где мы сочтем нужным, но не заставляет нас этого делать. Отделение имени ресурса от структуры принимаемой формы позволяет нам использовать одно логическое имя для поддержки нескольких типов взаимодействий. Мы можем обновить реализацию, не нарушая работы существующих клиентов. Скажем, при переходе от модели, в которой все существующие клиенты отправляют сообщения операцией POST в версии 1 схемы сообщений, мы можем добавить в реализацию поддержку версии 2 схемы, но при этом бизнес-система будет продолжать работать, как прежде (если это имеет смысл, конечно). Если мы когда-либо захотим избавиться от старой схемы, это можно сделать, но опять же в тот момент, когда мы сочтем нужным. Гибкость – одна из причин, по которым ресурсно-ориентированные архитектуры помогают вернуть главенствующее положение бизнесу: изменения в реализации не требуют обязательного обновления интерфейсной части. Упаковав унаследованную систему в REST-интерфейс, можно использовать ее и далее до тех пор, пока не появятся убедительные экономические причины для ее изменения. Конечно, существуют и другие технологии, которые позволяют инкапсулировать унаследованные системы подобным образом. В данном случае важен общий принцип использования логических имен, который помогает избежать нежелательных изменений на промежуточном уровне.

В своем стремлении к обеспечению горизонтальной масштабируемости REST-архитектура требует, чтобы запросы не имели состояния. Это означает, что любая информация, необходимая для ответа на запрос, должна содержаться в самом запросе. В этом случае система

управления нагрузкой сможет распределять запросы на любом количестве серверов реализации. При повышении нагрузки к сети можно подключить дополнительное оборудование, и любой из новых серверов сможет получать и обрабатывать запросы. Хотя это архитектурное ограничение вводилось, прежде всего, по соображениям масштабируемости, применение запросов без состояния к семантике GET-запроса имеет другое важное следствие: возможность кэширования результатов произвольных запросов. Адрес отвечающей стороны (основная часть URL) вместе с полным состоянием запроса (иерархия URL плюс параметры запроса) образует составной ключ хеширования для итогового набора (например, запрос базы данных, применяющий преобразование к другому блоку данных, и т. д.) Преимущества кэширования обходятся не бесплатно, но представить себе среду, использующую потенциал кэширования, не так уж трудно. Одна из многих привлекательных сторон ресурсно-ориентированной среды[1] NetKernel заключается в том, что она глубоко и эффективно использует эту возможность для обеспечения «архитектурной мемоизации[2]» практически без каких-либо усилий с вашей стороны. Подробнее см. ниже в разделе «Практическое применение ресурсно-ориентированных архитектур».

При наличии общей схемы построения имен всех интересующих нас ресурсов и процесса обработки логических запросов, допускающего изменение формы ресурса со временем или с изменением контекста, мы располагаем почти всей инфраструктурой, необходимой для переворота в управлении информацией в масштабах организации. Остается последнее: возможность описания адресуемых ресурсов в метаданных. Здесь на помощь приходит RDF (Resource Description Framework). Эта спецификация, рекомендованная W3C, использует модель графа для неограниченного включения информации об именованных сущностях. Кто создал ресурс? Когда он был создан? Для чего? С чем он связан? Возможность ссылки по имени и адресации существующих данных, хранящихся в реляционной базе данных, позволяет описать любые нужные данные без преобразования их в новую форму. Данные обычно оставляются в их текущем месте и интегрируются на том уровне, где это имеет смысл.

В следующем листинге N3-выражения RDF используются для описания создателя, названия, даты закрепления авторского права и лицензии, связанной с ресурсом. В примере представлены описания трех терминов (terms) Dublin Core Metadata Initiative[3] и одного термина из сооб-

[1] *http://1060.org*

[2] *http://en.wikipedia.org/wiki/Memoization*

[3] *http://dublincore.org*

щества Creative Commons[1]. Мы можем свободно использовать термины из любых существующих номенклатур, а также создавать новые:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix cc: <http://creativecommons.org/ns/> .
<http://bosatsu.net/team/brian/index.html> dc:creator
  <http://purl.org/people/briansletten> .
<http://bosatsu.net/team/brian/index.html> dc:title
  "Brian Sletten's Homepage" .
<http://bosatsu.net/team/brian/index.html> dc:dateCopyrighted
  "2008-04-26T14:22Z" .
<http://bosatsu.net/team/brian/index.html> cc:license
  <http://creativecommons.org/licenses/by-nc/3.0/> .
```

Мы можем не только пользоваться любыми терминами по своему усмотрению, но и добавлять новые термины и отношения в любой момент в будущем, не влияя на существующие отношения. Бессхемный подход выглядит в высшей степени заманчиво для каждого, кому когда-либо доводилось изменять схемы XML или реляционных СУБД. Кроме того, представляемая им модель данных не только выживает перед лицом неизбежных социальных, процедурных и технологических изменений, но и принимает их.

RDF находится в хранилище триплетов (triplestore) или другой базе данных, где для обращения к нему с запросами используется SPARQL или другой аналогичный язык. Большинство контейнеров с соответствующими семантическими возможностями поддерживает хранение RDF и запросы. В качестве примеров можно привести Mulgara Semantic Store[2], the Sesame Engine[3], Talis Platform[4], и даже Oracle 10g и последующих версий. Для выбора узлов графа может использоваться механизм поиска по шаблону, поэтому мы можем обращаться к своим ресурсам с запросами типа «Кто создал этот URL?», «Показать все ресурсы, созданные Брайаном», или «Найти все материалы с лицензией Creative Commons, созданные за последние шесть месяцев». Понятия «быть созданным», «обладать лицензией» и т. д. выражаются в соответствующих номенклатурах, но легко транслируются для наших целей. Благодаря гибкости модели данных в сочетании с выразительностью языка запросов описание, поиск и активизация REST-служб реализуются относительно прямолинейно. Конечно, это намного удобнее,

---

1 *http://creativecommons.org/ns*

2 *http://mulgara.org*

3 *http://openrdf.org*

4 *http://talis.com*

чем попытки поиска и активизации служб с использованием выхолощенных и неповоротливых технологий вроде UDDI.

Вооружившись возможностью адресации произвольных ресурсов, их выборки в различных формах и межноменклатурного описания, мы готовы применять эти идеи в корпоративных масштабах. Далее описывается информационная архитектура, поддерживающая «серфинг» в паутинах данных по аналогии с «серфингом» в Паутине документов.

Ресурсно-ориентированные архитектуры

Для ресурсно-ориентированного стиля характерен процесс выдачи логических запросов к именованным ресурсам. Запросы интерпретируются неким ядром и преобразуются в физическое представление ресурса (страница HTML, форма XML, объект JSON и т. д.).

На рис. 5.4 изображен основной стиль взаимодействия в ресурсно-ориентированной архитектуре (ROA, Resource-Oriented Architecture). Ресурсно-ориентированное ядро присваивает логическому запросу имя, разрешает его и возвращает запрашивающей стороне в некоторой форме. Именованный ресурс обычно трансформируется в запрос к базе данных или некоторый фрагмент функциональности, обеспечивающий управление информацией (например, REST-службу). Для человека, интересующегося информацией, обычно несущественно, что именно ответит на его запрос – сервлет, рестлет[1], модуль NetKernel или другая адресуемая функциональность, способная интерпретировать запрос. За этим логическим шагом скрывается целый мир возможностей и технологических решений; лишние подробности клиенту по-

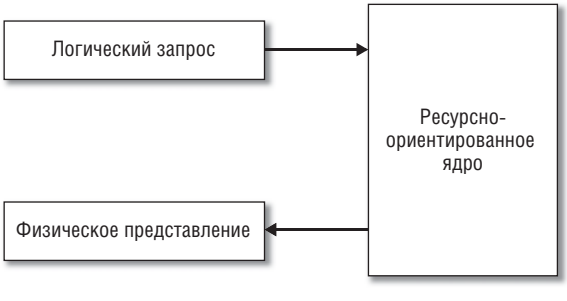


Рис. 5.4. *Ресурсно-ориентированные архитектуры*

[1] *http://restlet.org*

просту не сообщаются. Поддерживаются не все стили взаимодействий, но вас может удивить, сколько всего можно с комфортом спрятать за URL.

Возьмем адрес *http://server/getemployees&type=salaried*. Многие люди, полагающие, что они создают REST-архитектуру, создают URL похожего вида. К сожалению, это имя не является хорошим именем REST-службы (многие поклонники REST возразят, что оно вообще не соответствует канонам REST!), так как в нем смешиваются существительные и глаголы. Я обычно называю похожие конструкции «адресацией через URL» или «RPC через URL». В том, как REST отделяет существительные от глаголов, нет ничего сверхъестественного; просто такое разделение упрощает описание интересующих нас сущностей. Вышеупомянутый URL не удастся заново использовать для обновления списка работников, потому что отправка записи работника операцией POST по адресу */getemployees* выглядит совершенно бессмысленно. Если бы, скажем, URL-адрес имел вид *http://server/employee/salaried*, то выдача запроса GET привела бы к получению той же информации, но мы получаем долгоживущий адрес для представления бизнес-концепции «работников на ставке», тогда как адрес *http://server/employee/hourly* мог бы относиться к работникам с почасовой оплатой. Возможно, мы не станем обновлять эти информационные ресурсы, так как они представляют запросы к реальному хранилищу. Тем не менее, они следуют логике информационного пространства */employee*, по которому мы можем перемещаться и другими способами. Адрес *http://server/employee/12345678* представляет работника с конкретным идентификатором, а адрес *http://server/employee* может представлять всех работников. Отправка записи по последнему адресу операцией POST может представлять обновление записи работника после перевода, повышения и т. д. Операция DELETE по тому же адресу может означать, что ресурс с заданным именем более не представляет интереса для организации (например, из-за увольнения работника).

В этом проявляется одно из важнейших различий между REST и SOAP, которое иногда не понимают люди, путающие предназначения этих двух архитектурных стилей. Технология SOAP хорошо подходит для активизации поведения, но она не годится для управления информацией. Архитектура REST ориентирована на управление информацией, а не на активизацию поведения через URL. Если люди начинают чесать в затылке и прикидывать, достаточно ли четырех глаголов для всего, что они хотят сделать, скорее всего, они думают не об информации; их интересует активизация поведения.

Если вы применяете «RPC через URL», то с таким же успехом можете воспользоваться SOAP. Если вы рассматриваете важные бизнес-кон-

цепции как адресуемые информационные ресурсы, с которыми можно выполнять операции и представлять в разных формах в разных контекстах, вы выбираете REST; скорее всего, эта архитектура предоставит вам некоторые из преимуществ, характерных для Web. Даже если ваша подсистема реализации использует SOAP для обработки запросов, REST-интерфейс все равно может принести пользу. Такое строение адресов не только позволяет пользователям искать данные посредством «серфинга», но и создает потенциальную возможность кэширования результатов и устранения некоторых проблем с изменением контракта WSDL. Клиентский запрос проходит через логическую привязку и транслируется в сообщение SOAP. На сообщение генерируется ответ, содержимое которого может быть исключено из полученного результата. Нам просто не нужно оповещать всех о факте использования SOAP; этот процесс может открыть новые стратегии архитектурной миграции.

Как видно из рис. 5.5, один и тот же именованный ресурс может возвращаться в разных физических формах для разных контекстов. Нетрудно представить себе отчет компании, оформленный в виде инфор-

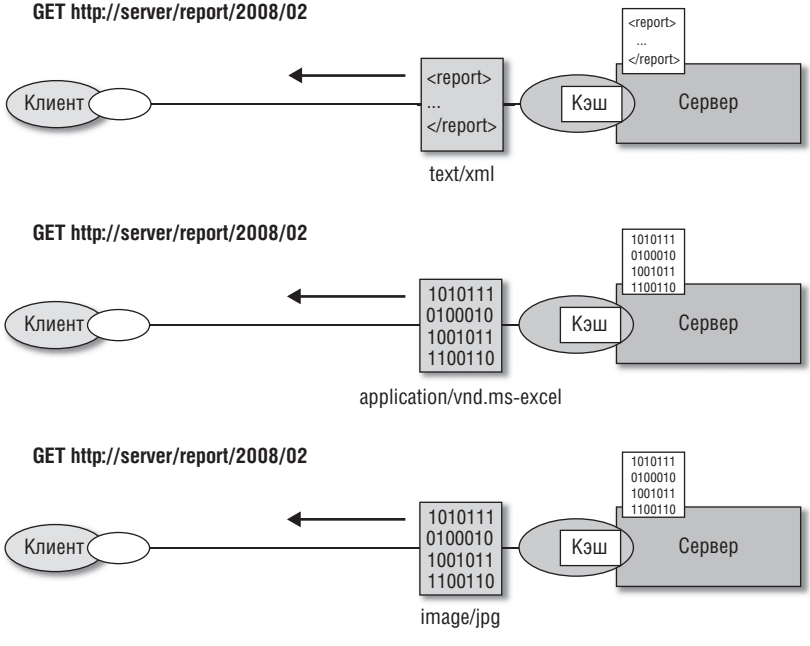


Рис. 5.5. *Согласование контента в ресурсно-ориентированной среде*

мационного пространства с возможностью перемещения по временной шкале (например, год и месяц). Пока существует только один тип отчета, имя *http://server/report/2008/02* останется достаточно хорошим, долгоживущим именем. Какой бы момент будущего ни наступил, отчет все равно будет относиться к февралю 2008 года. В одном сценарии мы можем обращаться к данным в формате XML, в другом – в формате электронной таблицы Excel, в третьем – в формате изображения JPEG для включения в сводный отчет. Нам не нужны разные имена для всех трех сценариев, поэтому выбор будет осуществляться на основании согласования контента. Ресурсно-ориентированное ядро должно знать, как отвечать на запросы определенного типа, но это делается достаточно просто. В будущем может появиться формат данных, не поддерживаемый ни одним из существующих клиентов. При этом изменять существующих клиентов не придется – поддержка будет добавлена на сервере, и ей воспользуется какой-то другой клиент. Такая гибкость по отношению к изменениям была заложена в архитектуру Web, и нам хотелось бы, чтобы она была в равной степени присуща корпоративным системам. Клиент и сервер могут согласовать определенную форму именованного ресурса в ходе разрешения запроса. Таким образом, один именованный ресурс может иметь разную структуру в зависимости от контекста (XML на среднем уровне, JSON в броузере и т. д.) При желании сервер может кэшировать структурированные формы.

Кроме выбора физического представления в контексте разрешения запроса мы также можем поручить серверу принимать решение о том, какую часть набора данных следует возвращать в зависимости от личности пользователя, используемого приложения и т. д. Представьте себе сценарий, в котором оператор центра обработки звонков, использующий соответствующее приложение, для устранения проблемы должен обратиться к конфиденциальной информации: номерам социального страхования, номерам кредитных карт (вероятно, только последним четырем цифрам), домашним адресам и т. д. В данной ситуации обращение оператора к информации объясняется конкретной деловой необходимостью. С другой стороны, если тот же работник использует другое приложение в другом контексте (например, пакет маркетингового анализа), у него вряд ли имеется реальная необходимость в обращении к конфиденциальной информации, хотя запрос к данным того же клиента можно разрешить для обращения к демографическим данным и истории покупок. В этом случае контекст не требует доступа к конфиденциальным данным, и мы можем реализовать автоматическую фильтрацию для исключения или шифрования защищенной информации. Выбор зависит от того, как данные будут использоваться дальше. Шифрование потребует доступа к ключам, что создает допол-

нительные хлопоты. С таким же успехом можно убрать конфиденциальные данные, чтобы включить их в другом контексте в случае необходимости.

Одноточечное управление доступом не создает серьезных проблем в традиционных корпоративных архитектурах. Однако с учетом распространения потоков операций, явного моделирования бизнес-процессов и т. п. нетрудно представить, как у пользователя приложения возникает необходимость активизации некоторой функции или возможности в нескольких контекстах. Если между системами передаются фактические данные, то разработчики приложений несут ответственность за проблемы управления доступом при пересечении границ приложений. Если вместо этого передать ссылку на данные, то ответственность снимается с исходного приложения, и на вас начинает работать централизованный механизм управления доступом. Многие существующие системы SOA ограничивают доступ к службам в зависимости от личностей или ролей, но они редко способны ограничивать конкретные данные, проходящие через эти службы. Этот недостаток – одна из причин, по которым традиционные веб-службы считаются запутанными и недостаточно безопасными. Политика доступа должна применяться как к поведению, так и к данным в контексте, но без возможности присваивания имени данным и ограничения контекста это сделать очень трудно.

При первом знакомстве с ресурсно-ориентированными архитектурами люди обычно беспокоятся по поводу предоставления доступа к конфиденциальной информации по ссылкам. Возвращение блоков данных за непрозрачными запросами кажется им более безопасным. Им трудно отделить акт идентификации и разрешения ссылок от контекста, в котором оно выполняется. Контекст содержит достаточно информации для принятия решения о том, стоит ли передавать информацию определенному пользователю. Он ортогонален к самому запросу, а его требования удовлетворяются при помощи систем аутентификации и авторизации, действующих в организации. Для защиты данных могут привлекаться любые механизмы, от базовой аутентификации HTTP до IBM Tivoli Access Manager и OpenID и других интегрированных систем проверки. Мы можем следить за тем, кто имеет доступ к той или иной информации, и шифровать транспортный поток с применением одно- или двустороннего SSL для предотвращения перехвата. Адресуемость не равносильна уязвимости. Более того, передача ссылок вместо данных является более безопасной, лучше масштабируемой стратегией. Ресурсно-ориентированный стиль не снижает уровня защиты, хотя он и не отягощен усложняющими средствами безопасности (XML Encryption, XML Signature, XKMS, XACML, WS-Security, WS-Trust, XrML и т. д.) – скорее

он обеспечивает более надежную защиту, потому что позволяет понять модель угрозы и способы применения стратегий защиты.

Важность этих идей наглядно проявляется тогда, когда мы сталкиваемся с непростыми и очень серьезными реалиями соблюдения нормативных ограничений. Компании, занимающиеся выпуском кредитных карт, организации по надзору за здравоохранением, аудиторы корпоративного управления и т. д. – все они могут создать серьезные неприятности, требуя доказательств того, что доступ к конфиденциальной информации предоставляется только тем, кому он необходим по работе. Даже если в вашей организации все ограничения выполняются, если это трудно доказать («Сначала загляните в системный журнал, потом отследите переход сообщения между посредниками, где оно преобразуется в запрос, как видно из другого журнала...»), процесс может быть довольно дорогостоящим. Применение декларативной политики управления доступом к разрешению логических ссылок позволяет в явной (и легко проверяемой) форме определить, кому, когда и к какой информации предоставляется доступ.

Приложения, управляемые данными

Если организация пошла на хлопоты с обеспечением адресации своих данных, выигрыш от такого решения не ограничивается возможностью кэширования результатов подсистемами выборки данных и неразрушающим внедрением новых технологий. А именно, у нас появляется возможность введения совершенно новых классов приложений, управляемых данными, и стратегий интеграции. Присваивание имен данным и возможность запрашивать их способом, удобным для приложения, обеспечивает уровень сбора и обработки деловой информации, который приведет в восторг большинство аналитиков. В ходе Simile[1], совместного проекта W3C и группы MIT CSAIL, была проделана огромная работа, отлично демонстрирующая эти идеи и причины восторга.

Допустим, вы хотите проанализировать влияние различных маркетинговых стратегий на трафик веб-сайта и продажи. Необходимая информация берется из электронной таблицы, базы данных и нескольких журнальных файлов и отчетов, построенных программами веб-аналитики. Хотя объединение этих источников информации нельзя назвать высшей математикой, поиск, выборка, преобразование и повторная публикация результатов потребуют нетривиальных усилий.

[1] *http://simile.mit.edu*

Если просто создать сводный отчет в формате электронной таблицы и отослать его по электронной почте, фактически мы потеряем возможность получения результатов в будущем без поиска по нашим загроможденным почтовым ящикам. Использование CMS или другой системы управления документами увеличит время, необходимое для получения результата. Независимо от частоты построения отчетов процесс придется каждый раз повторять заново.

В ресурсно-ориентированной архитектуре мы можем просто адресовать источник каждого элемента данных и запрашивать их в формате файлов JSON, чтобы их можно было легко использовать в броузерной среде. Проект Exhibit[1] из Simile с представлением Timeline[2] практически предоставляет такую возможность. Немного поработав над преобразованием таблиц Excel в объекты JSON, вы получите пригодную для повторного использования среду, которая позволяет строить и публиковать маркетинговые отчеты за считанные секунды. А теперь представьте, как та же инфраструктура с такой же легкостью обеспечивает выборку других типов данных для разных форм анализа и построения отчетов, и вы начнете понимать полезность Паутины адресуемых данных. Такие среды постепенно начинают появляться в корпоративных сетях; если ваша организация еще не может с такой же легкостью связать свои данные воедино, ей стоит позаботиться об этом.

Практическое применение ресурсно-ориентированных архитектур

Недавно я построил ресурсно-ориентированную систему в ходе переработки архитектуры, выполняемой моей компанией для системы PURL (Persistent URL). Исходная реализация PURL[3] была написана около 15 лет назад. Она представляла собой ответвление Apache 1.0, написанное на C; на тот момент времени код был весьма передовым[4]. Код PURL долго оставался надежным фрагментом инфраструктуры Интернета, но со временем он начал устаревать и нуждаться в модернизации, особенно в отношении поддержки предложенной W3C TAG «рекомендации 303» и повысившейся нагрузки. Доступ к большей части данных осуществлялся с веб-страниц или при помощи специально

---

1 *http://simile.mit.edu/exhibit*

2 *http://simile.mit.edu/timeline*

3 *http://purl.org*

4 Кодовая база была заложена в основу чрезвычайно успешного сервиса TinyURL (*http://tinyurl.com*).

сгенерированных сценариев CGI-bin, так как в то время броузер считался единственным реальным клиентом. Когда мы начали понимать возможности применения долгосрочных, однозначных идентификаторов в Семантической Паутине, биологии, издательском деле и других аналогичных сферах, стало ясно, что пришло время переосмыслить архитектуру, и сделать ее более удобной как для людей, так и для программ.

Система PURL проектировалась для того, чтобы сократить разрыв между удобными и успешно разрешаемыми именами. Каждый, кто когда-либо публиковал контент в Web, знает, что при перемещении контента ссылки нарушаются. Концепция PURL определяет хорошо запоминающееся, логичное имя, соответствующее некоторому адресу. Например, можно определить PURL-адрес, который связывает *http://purl.org/people/briansletten* с *http://bosatsu.net/foaf/brian.rdf*, и возвращает код 303 – признак ответа «см. также». Я сам не являюсь ресурсом с сетевой адресацией, но дополнительную информацию обо мне можно найти в моем файле FOAF[1] (Friend-of-a-Friend). Я могу сообщить этот PURL-адрес любому, кто захочет создать ссылку на мой файл FOAF. Если я когда-нибудь перейду в другую компанию, я могу обновить PURL в соответствии с новым реальным адресом своего файла FOAF. Все существующие ссылки останутся действительными; они просто будут возвращать код 303 для нового местоположения. Процесс показан на рис. 5.6. Сервер PURL реализует рекомендацию W3C TAG (Technical Architecture Group), согласно которой код ответа 303 может использоваться для получения дополнительной информации о ресурсах, не имеющих сетевой адресации.

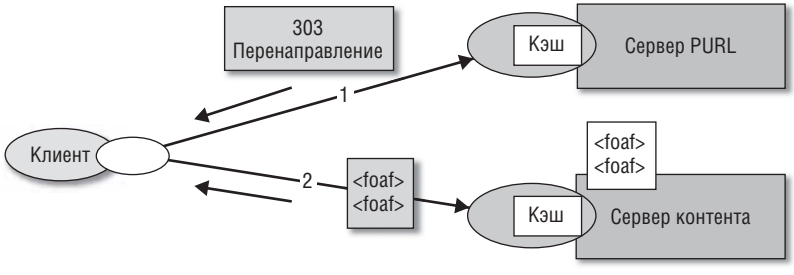


Рис. 5.6. *Перенаправление «см. также» с использованием PURL*

[1] *http://foaf-project.org*

Помимо поддержки перенаправления каждый важный блок данных в системе PURL должен представлять собой адресуемый информационный ресурс. Это не только упрощает взаимодействие с пользовательским интерфейсом, но и выводит возможности повторного использования данных за изначально запланированные рамки. Для манипуляций с ресурсами необходимы полномочия владельца, однако любой желающий сможет получить определение PURL. Существует как прямой процесс разрешения обращений по PURL вида *http://purl.org/employee/briansletten* (что приведет к перенаправлению 303), так и косвенный REST-адрес ресурса PURL *http://purl.org/admin/purl/employee/briansletten*, который вернет определение PURL следующего вида:

```
<purl status="1">
    <id>/employee/briansletten</id>
    <type>303</type>
    <maintainers>
       <uid>brian</uid>
    </maintainers>
    <seealso>
       <url>http://bosatsu.net/foaf/brian.rdf</url>
    </seealso>
</purl>
```

Клиенты сервера PURL могут перейти к определению данных как к средству получения информации о ресурсе PURL без его фактического разрешения. Для получения этой информации не нужно писать специальный код: его можно просмотреть в броузере или сохранить из командной строки при помощи *curl*. Соответственно можно представить себе сценарии командного процессора, которые по данным из наших информационных ресурсов определяют, ссылается ли PURL на действительный ресурс, и возвращают разумный результат. Если нет, мы можем найти владельца PURL и отослать сообщение на соответствующий адрес электронной почты. Адресуемые данные быстро проникают в сценарии, приложения и виджеты, даже если это и не было предусмотрено исходными планами, потому что это просто и удобно.

В интересах полной открытости мы не стали поддерживать формат запросов JSON в исходной версии, что привело к усложнению пользовательского интерфейса AJAX. Работа с XML в JavaScript оставляет желать много лучшего. Даже несмотря на то, что во внутренней реализации использовался формат XML, нам следовало не жалеть усилий на поддержку формата JSON для разбора в броузере. Конечно, упущение будет вскоре исправлено, но я хотел лишний раз подчеркнуть те преимущества, которые бы принесла изначально правильная реализация.

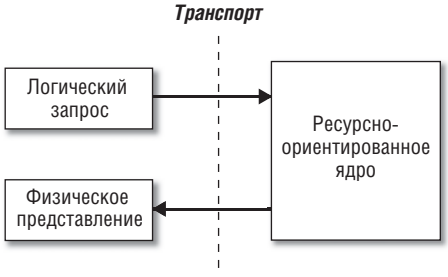


Рис. 5.7. *Упрощенная ресурсно-ориентированная архитектура*

Возможно, поддерживать сразу все форматы данных с самого начала не обязательно, но в наши дни поддержка XML и JSON является хорошей отправной точкой.

Попутно отметим, что для представления описанной архитектуры можно было выбрать разные контейнеры и инструменты. Любая программа, отвечающая на запросы HTTP, может выполнять функции сервера PURL. Мы приходим к упрощенному, но полезному воплощению REST-совместимых интерфейсов и ресурсно-ориентированных архитектур, показанному на рис. 5.7. Любой веб-сервер или сервер приложений может стать упрощенным ресурсно-ориентированным ядром. Логические запросы HTTP интерпретируются как запросы к сервлетам, рестлетам или другой аналогичной адресуемой функциональности.

Мы решили заложить NetKernel в основу этой архитектуры, потому что NetKernel является воплощением ресурсно-ориентированных архитектур и имеет двойную лицензию, разрешающую применение как в коммерческих проектах, так и в проектах с открытым кодом. Идея логической привязки между уровнями с разными представлениями встроена в архитектуру продукта и обеспечивает аналогичные преимущества в отношении гибкости, масштабируемости и простоты. Уровни связываются при помощи асинхронно разрешаемых логических имен. Более глубокое воплощение ресурсно-ориентированных архитектур выглядит примерно так, как показано на рис. 5.8. Программная инфраструктура NetKernel интересна тем, что идея логического связывания ресурсов переведена в ее внутреннюю структуру, так что логические запросы HTTP могут преобразовываться в другие логические запросы. Эта архитектура отражает свойства Web в программной среде времени выполнения.

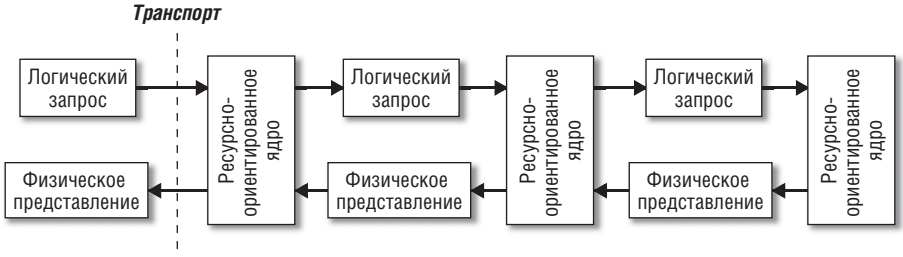


Рис. 5.8. *Глубокая ресурсно-ориентированная архитектура*

Внешнему URL-адресу *http://purl.org/employee/briansletten* через механизм *rewrite*[1] (*перезапись*) ставится в соответствие функциональный блок, называемый *аксессором*[2] (*accessor*). Аксессоры находятся в модулях, экспортирующих общедоступные определения URI с представлением адресного пространства, которому они будут отвечать. Такая схема удобна тем, что она позволяет радикально изменить технологии реализации в новой версии модуля, а затем просто переписать правила замены так, чтобы они ссылались на новую реализацию. Для клиента это останется незаметным при условии возвращения совместимых ответов. В современных объектно-ориентированных языках такая гибкость может имитироваться при помощи интерфейсов, но при этом все равно остается «физическая» привязка к определению интерфейса. При наличии только логической привязки мы обязаны поддерживать ожидания существующих клиентов, но в остальном ничто не связывает нас ни с какой конкретной подробностью реализации. Полезные качества, характерные для взаимодействий через URI в Web, достигаются в локальном программном продукте!

Во внутренней реализации используется паттерн «Команда[3]», связывающий тип запроса с реализацией. Метод HTTP GET связывается с функцией `GetResourceCommand`, не сохраняющей состояния. При поступлении запроса мы извлекаем команду из ассоциативного массива и передаем ей запрос. Отсутствие состояния в архитектуре REST гарантирует, что вся информация, необходимая для ответа на запрос, содержится

---

1 *http://en.wikipedia.org/wiki/Rewrite_engine* – механизм изменения представления URL-адреса, обычно используется для приведения его к более удобочитаемому и краткому виду. – *Примеч. ред.*

2 *http://docs.1060.org/docs/3.3.0/book/gettingstarted/doc_intro_code_accessor.html*

3 *http://en.wikipedia.org/wiki/Command_pattern*

в самом запросе, поэтому нам не нужно поддерживать состояние в экземпляре команды. Мы можем обратиться к состоянию запроса через экземпляр `context` в приведенном ниже коде. Для Java-программиста этот код выглядит относительно тривиально. Вызовы объектов Java, перехват исключений – словом, обычное дело. Обратите внимание на использование интерфейса `IURAspect`. Фактически мы говорим, что нас не интересует конкретный формат ресурса. Это может быть экземпляр DOM, экземпляр JDOM, строка, массив байтов; для наших целей это несущественно. Прежде чем отвечать на запрос, инфраструктура преобразует его в поток байтов с метаданными. Если бы мы хотели получить ресурс в конкретной форме, поддерживаемой инфраструктурой, то мы бы просто запросили его в этой форме. Декларативный, ресурсно-ориентированный подход помогает радикально сократить объем кода, необходимого для обработки данных, а также позволяет выбрать правильный инструмент для выполняемой операции:

```
if(resStorage.resourceExists(context, uriResolver)) {
    IURAspect asp = resStorage.getResource(context, uriResolver);
    // Отфильтровать ответ, если имеется фильтр
    if (filter!=null) {
        asp = filter.filter(context, asp);
    }
    // Стандартный код ответа 200 подходит
    IURRepresentation rep = NKHelper.setResponseCode(context, asp, 200);
    rep = NKHelper.attachGoldenThread(context, "gt:" + path , rep);
    retValue = context.createResponseFrom(rep);
    retValue.setCacheable();
    retValue.setMimeType(NKHelper.MIME_XML);
} else {
    IURRepresentation rep = NKHelper.setResponseCode(context,
        new StringAspect("No such resource: " +
                                uriResolver.getDisplayName(path)), 404);
    retValue = context.createResponseFrom(rep);
    retValue.setMimeType(NKHelper.MIME_TEXT);
}
```

Большая часть информационных ресурсов возвращает код 200 для запросов GET. Разумеется, PURL переопределяют это поведение для возвращения кодов 302, 303, 307, 404 и т. д. Интересный ресурсно-ориентированный аспект обнаруживается при изучении PURL-ориентированной реализации метода `resStorage.getResource()`:

```
INKFRequest req = context.createSubRequest("active:purl-storage-query-purl");
req.addArgument("uri", uri);
IURRepresentation res = context.issueSubRequest(req);
return context.transrept(res, IAspectXDA.class);
```

По сути, мы выдаем логический запрос к URI `active:purl-storage-query-purl` с аргументом `ffcpl:/purl/employee/briansletten`. Не обращайте внимания на необычную схему URI; она используется для представления внутренних запросов NetKernel. Мы не знаем, какой код будет реально использован для выборки PURL в запрашиваемой форме, да это и не важно. В ресурсно-ориентированной среде мы просто говорим: «Код, который отвечает по этому URI, сгенерирует ответ за меня». Теперь мы можем быстро организовать раздачу статических файлов клиентам модуля в ходе проектирования, а потом построить что-то вроде отображения на реляционную базу данных на основе Hibernate. Переход реализуется заменой части, отвечающей на URI `active:purl-storage-query-purl`. Клиентский код может ничего не знать об изменениях. Если переместить процесс разрешения PURL с локального уровня долгосрочного хранения на уровень удаленной выборки, для клиентского кода все равно ничего не изменится. Все эти преимущества уже обсуждались нами в более общем контексте ресурсно-ориентированной корпоративной обработки данных, реализованной в мощной программной среде.

Помимо слабой привязки уровней мы также пользуемся преимуществами идемпотентных запросов без состояния. Приведенный ранее фрагмент кода, получающий определение PURL, во внутреннем представлении преобразуется в асинхронно планируемый запрос к URI `active:purl-storage-query-purl+uri@ffcpl:/purl/employee/briansletten`. Как упоминалось ранее, он становится составным ключом хеширования, представляющего результат запроса к уровню долгосрочного хранения. Даже притом, что мы ничего не знаем о коде, которому передается управление, NetKernel все равно может кэшировать результат. Мы имеем дело с «архитектурной мемоизацией», о которой я упоминал ранее. У реального процесса чуть больше нюансов, но в целом происходит следующее. Если кто-то пытается разрешить уже использовавшийся PURL (через внутренний запрос или через HTTP REST-интерфейс), то мы можем извлечь результат из кэша. Вероятно, это не произведет особого впечатления на тех, кто встраивал кэширование в свои веб-страницы, но если заглянуть поглубже, это весьма многообещающий результат. Так может кэшироваться любой потенциальный запрос к URI независимо от того, читаем ли мы файлы с диска, загружаем их через HTTP, трансформируем документ XML по таблице XSLT или вычисляем «пи» с точностью до 10 000 цифр. В каждом из этих запросов используется логический, лишенный состояния, асинхронный результат, и в каждом случае открывается возможность для кэширования. В результате применения ресурсно-ориентированного архитектурного стиля создается программный продукт, который хорошо мас-

штабируется, эффективно работает, поддерживает кэширование и работает с использованием унифицированных, логичных интерфейсов. Это приводит к созданию более надежных, более гибких архитектур, которые хорошо масштабируются, – как Web и по тем же причинам.

Заключение

Использование ресурсно-ориентированных архитектур сопряжено с определенным риском. С одной стороны, новичкам этот подход может показаться немного странным и непроверенным. Люди, беспокоящиеся о своих резюме, предпочитают придерживаться опробованных и надежных решений. С другой стороны, людям, изучавшим Web и основные структурные элементы этой среды, он кажется абсолютно логичным и представляет самую крупную, самую успешную сетевую программную архитектуру, когда-либо спланированную и реализованную на практике. С другой точки зрения этот подход предоставляет мощный механизм инкапсуляции и повторного использования существующего кода, сервисных функций и инфраструктур через интерфейсы, обладающие логичными именами и не выдающие подробностей реализации для многих форм взаимодействий. Мы получаем большую свободу выбора запросов, принимаемых сервером, без нарушения работы существующих клиентов. С течением времени на сервере может появляться поддержка новых структурных форм существующих данных. Становится возможным изменение внутренних реализаций, не влияющее на работу клиентов. Кроме того, этот архитектурный стиль обеспечивает такие важные свойства, как масштабирование, кэширование, управление доступом в зависимости от информации и простой контроль за соблюдением нормативных ограничений.

Программисты обычно не интересуются данными; они интересуются алгоритмами, объектами и другими подобными конструкциями. У нас имеются довольно конкретные рекомендованные планы и технологии для архитектур на базе J2EE, .NET и SOAP. К сожалению, многие из этих планов не рассматривают информацию как полноправную сторону архитектуры. Они ограничивают нас привязками, которые затрудняют внесение изменений без нарушения работы существующих клиентов. Мы пребывали в этом тупике уже много лет, и бизнесу надоело платить за него. Предполагалось, что веб-службы станут хорошей стратегией выхода, но из-за неуместных уровней абстракции и излишней сложности этой технологии проектировщики разочаровались в ней. Настало время отойти от программно-центрических архитектур и сосредоточиться на информации и ее перемещениях. Мы по-прежнему пишем свои программы при помощи хорошо известных и любимых

нами инструментов; просто программы уходят с первого плана в наших архитектурах.

Ресурсно-ориентированный подход налаживает связи между экономическими подразделениями организаций и техническими отделами, поддерживающими их работу. Информационно-центрические представления о связях в наших системах приносят реальную пользу. Вместо того чтобы переписывать все заново после очередной Великой Идеи наших поставщиков ПО, мы можем извлечь ценные уроки из Web и важных свойств, обусловленных особенностями архитектурного стиля этой среды. Архитектура – это обитаемая скульптура; последствия принятых решений действуют в течение довольно долгого времени. Воспользуйтесь представившейся возможностью и наделите свои архитектуры функциональностью, красотой и гибкостью, чтобы ваше взаимодействие с ними стало более полезным и приятным.

Принципы и свойства	Структуры
✓ Гибкость	✓ Модуль
Концептуальная целостность	✓ Зависимость
✓ Возможность независимого изменения	Обработка
✓ Автоматическое распространение	✓ Доступ к данным
Удобство построения	
✓ Адаптация к росту	
✓ Сопротивление энтропии	

6

Архитектура Facebook Platform

Дэйв Феттерман

Покажите мне блок-схемы, не показывая таблиц, и я останусь в заблуждении. Покажите мне ваши таблицы, и блок-схемы, скорее всего, не понадобятся: они будут очевидны.

Фред Брукс «Мифический человеко-месяц»

Введение

Многие студенты, изучающие информатику, полагают, что Фред Брукс в этой цитате имел в виду: «Покажите мне свой код, не показывая структур данных...». Специалисты по информационным архитектурам хорошо понимают, что в большинстве систем центральное место занимают данные, а не алгоритмы. А с ростом влияния Web данные, производимые и потребляемые пользователем, как никогда стимулируют применение информационных технологий. Пользователи Web не ищут по ссылкам алгоритм быстрой сортировки. Они посещают хранилище данных.

Эти данные могут быть общедоступными, как телефонный справочник; закрытыми, как интернет-магазин; персонализированными, как блог; открытыми, как прогноз погоды; тщательно охраняемыми, как данные клиентов на сайте банка. В любом случае функциональность практически любого веб-присутствия, обращенная к пользователю, сводится к предоставлению интерфейса к данным, специфическим для конкретного сайта. Информация образует реальную ценность практически любого сайта, построенного исследовательской группой профессионалов или созданного рядовыми пользователями по всему миру. Благодаря данным создаются продукты, пользующиеся успехом у пользователей, поэтому архитекторы строят на их основе остальные компоненты традиционного «n-уровневого» программного стека (*логику* и *представление*).

В этой главе рассказано о данных Facebook и их эволюции с созданием платформы Facebook Platform.

Facebook (*http://facebook.com*) является примером архитектуры, построенной на основе социальных данных, предоставленных пользователем: сведений о личных отношениях, биографической информации, текстового и другого контента. Инженеры Facebook строили архитектуру сайта с расчетом на отображение и обработку этих социальных данных. От социальных данных непосредственно зависит большая часть бизнес-логики сайта: последовательность и закономерности доступа к разным страницам, реализация поиска, отображение контента из ленты новостей, применение правил видимости контента. С точки зрения пользователя ценность сайта напрямую определяется ценностью данных, внесенных в систему им самим и теми, с кем он общается.

На концептуальном уровне «социальный сайт Facebook» представляет собой стандартный n-уровневый стек, в котором пользовательский запрос извлекает из внутренних библиотек Facebook данные, преобразуемые по логике Facebook и предназначенные для отображения средствами Facebook. Затем проектировщики Facebook осознали, что эти данные могут приносить пользу и за рамками их контейнера. Создание Facebook Platform очевидно изменило целостный образ системы доступа к данным Facebook; оно ознаменовало переход на перспективу значительно более широкую, чем изолированная функциональность n-уровневого стека, с целью интеграции с внешними системами в форме приложений. С социальными данными пользователей, заложенными в основу архитектуры, развитие платформы привело к появлению семейства веб-служб (Facebook Platform Application Programming Interface, или Facebook API), языка запросов (Facebook Query Language, или FQL) и управляемого данными языка разметки (Facebook Markup

Language, или FBML), призванных объединить системы прикладных разработчиков с системами Facebook.

Наборы данных получают все более широкое распространение, а пользователи все чаще желают унифицированного использования своих данных среди разных веб- и настольных продуктов. Архитектор, читающий эту главу, скорее всего является либо потребителем такой платформы, либо производителем аналогичной платформы для данных его собственного сайта. В этой главе вы узнаете, как проходило контролируемое открытие данных Facebook для внешних стеков, какие архитектурные решения следовали из каждого шага эволюции данных и как они согласовывались с особыми требованиями конфиденциальности, присущими социальным сетям. В частности, рассматриваются следующие вопросы:

- Ситуации, в которых такая интеграция приносит пользу.
- Перемещение функций данных из внутреннего стека в доступные извне веб-службы (Facebook API).
- Авторизация доступа к веб-службе с учетом сохранения конфиденциальности социальной системы.
- Создания языка запросов к данным, упрощающего использование веб-службы новыми клиентами (Facebook FQL).
- Создание управляемого данными языка разметки – как для интеграции выходных данных приложения обратно в Facebook, так и для обеспечения возможности использования недоступных иным образом данных (Facebook FBML).

И после значительной эволюции архитектуры приложения в сравнении с отдельным стеком:

- Создание технологий, устраняющих различия между опытом работы пользователя с Facebook и опытом работы с внешними приложениями.

Для потребителей платформ данных в этой главе описаны принятые архитектурные решения и их обоснования. Такие темы, как пользовательские сеансы и аутентификация, веб-службы, различные способы управления логикой приложения, будут постоянно встречаться на платформах такого рода в Web. Понимание заложенных в них идей расширит ваш кругозор в области архитектуры данных; кроме того, оно будет полезно для прогнозирования новых возможностей и форм, которые могут быть созданы разработчиками этих платформ в будущем.

Производителю платформы данных следует помнить о своем наборе данных и учиться на примере открытия модели данных Facebook. Некоторые архитектурные решения специфичны для Facebook или, по

крайней мере, для социальных данных, защищенных требованиями конфиденциальности, и могут быть не в полной мере применимы к произвольному набору данных. Тем не менее, на каждом шаге будет приводиться описание проблемы, управляемое данными решение и его высокоуровневая реализация. С каждым новым решением фактически создается новый продукт или платформа, поэтому в каждой точке необходимо проверить новый продукт на соответствие ожиданиям пользователей. Мы последовательно создаем новые технологии, сопровождающие каждый шаг эволюции, а иногда изменяем веб-архитектуру, окружающую само приложение.

Версия Facebook Platform с открытым исходным кодом доступна по адресу *http://developers.facebook.com/*. Как и большая часть этого кода, примеры данной главы написаны на PHP. Попробуйте разобраться в них (учтите, что код примеров был сокращен для ясности).

Для начала мы попробуем разобраться, когда могут пригодиться подобные интеграции. В нашем первом примере рассматривается «внешняя» логика приложения и данные (книжный магазин), социальные данные Facebook (информация о пользователях и «дружеские» связи), и возможные причины для интеграции одного с другим.

Основные данные внешнего приложения

Веб-приложения – даже те, которые не являются производителями или потребителями тех или иных платформ данных, – в значительной мере зависят от своих внутренних данных. Для примера возьмем *http://fettermansbooks.com* – гипотетический сайт, предоставляющий информацию о книгах (и, скорее всего, возможность купить эти книги). На сайте может быть представлен список книг с возможностью поиска, основная информация по каждой книге и даже отзывы пользователей. Доступ к этой информации формирует основу функциональности приложения и служит причиной для всей остальной архитектуры. Сайт может использовать Flash и AJAX, он может быть доступен с мобильных устройств, может обладать великолепным интерфейсом... Но по сути *http://fettermansbooks.com* существует главным образом для того, чтобы предоставить посетителям доступ к информации по основным отображениям данных, показанным в листинге 6.1.

Листинг 6.1. Информационные отображения в примере с книгами

```
book_get_info : isbn -> {title, author, publisher, price, cover picture}
book_get_reviews: isbn -> set(review_ids)
bookuser_get_reviews: books_user_id -> set(review_ids)
review_get_info: review_id -> {isbn, books_user_id, rating, commentary}
```

Все эти отображения в конечном итоге реализуются в форме, очень похожей на простую выборку из индексированной таблицы данных. Любой достойный книжный сайт, скорее всего, реализует и другие, не столь простые функции – например, элементарный «поиск» их листинга 6.2.

Листинг 6.2. Отображение простого поиска

```
search_title_string: title_string -> set({isbn, relevance score})
```

Для каждого ключа в области определения этих функций обычно создается как минимум одна веб-страница сайта *http://fettermansbooks.com* – уникальная логика, относящаяся к диапазону данных, отображается по уникальному пути. Например, для просмотра всех рецензий автора X пользователь сайта *http://fettermansbooks.com* направляется на страницу вида *fettermansbooks.com/reviews.php?books_user_id=X*, а для просмотра всей информации о конкретной книге с кодом ISBN Y (со ссылками на страницы отзывов) будет открыта страница *http://fettermansbooks.com/book.php?isbn=Y*.

У таких сайтов, как *http://fettermansbooks.com*, есть одно важное свойство: практически любые данные доступны любому пользователю. Сайт генерирует весь контент, скажем, по отображению *book_get_info*, чтобы предоставить пользователю как можно более полную информацию о книге. Возможно, для сайта, продающего книги, такая стратегия оптимальна, но в приводимых далее примерах с социальными данными ограниченная видимость информации определяет многие архитектурные решения на уровне доступа к данным.

Основные данные Facebook

С ростом популярности семейства технологий, называемого Web 2.0, центральная роль данных в системах стала только более очевидной. Важнейшие особенности реализованных технологий Web 2.0 заключаются в том, что они управляются данными, причем большая часть этих данных предоставляется самими пользователями.

Facebook, как и *http://fettermansbooks.com*, определяет ряд первичных отображений данных, определяющих общее впечатление и функциональность сайта. Очень сильно сокращенный набор этих отображений Facebook представлен в листинге 6.3.

Листинг 6.3. Примеры отображений социальных данных

```
user_get_friends: uid -> set(uids)
user_get_info: uid -> {name, pic, books, current_location,...}
can_see: {uid_viewer, uid_viewee, table_name, field_name} -> 0 or 1
```

Здесь `uid` обозначает (числовой) идентификатор пользователя Facebook, а информация, возвращаемая `user_get_info`, относится к контенту из профиля пользователя (см. описание `users.getInfo` в документации разработчика Facebook) – дополненного названиями любимых книг пользователя, введенных на сайте *http://facebook.com*. По сути, эта система не так уж сильно отличается от *http://fettermansbooks.com*, если не считать того, что данные (а, следовательно, и функциональность сайта) концентрируются на связях пользователя с другими пользователями («друзья»), контенте пользователя («информация профиля») и правилах видимости контента (`can_see`).

Набор данных `can_see` специфичен. В системе Facebook существует основополагающее представление о конфиденциальности данных, сгенерированных пользователем, – бизнес-правила, определяющие возможность просмотра пользователем X информации пользователя Y. Эти данные никогда не открываются напрямую, но они определяют очень важные факторы, которые будут встречаться нам снова и снова при изучении примеров интеграции внешней логики и данных с логикой и данными Facebook. Повсеместное использование этого набора данных в Facebook отличает эту систему от таких сайтов, как *http://fettermansbooks.com*.

Существование Facebook Platform и других социальных платформ подтверждает полезность социальных отображений такого типа – как внутри сайта *http://facebook.com*, так и при интеграции с функциональностью внешнего сайта (такого как *http://fettermansbooks.com*).

Платформа приложений Facebook

С точки зрения пользователя обоих сайтов, *http://fettermansbooks.com* и *http://facebook.com*, структура интернет-приложений на этой стадии выглядит примерно так, как показано на рис. 6.1.

В обычной n-уровневой архитектуре приложение отображает входные данные (для Web – совокупность информации GET, POST и cookie) на запросы к физическим данным, которые обычно хранятся в базе данных. Они преобразуются в данные, находящиеся в памяти, и передаются *бизнес-логике* для обработки. Выходной модуль преобразует объекты данных в выходные форматы HTML, JavaScript, CSS и т. д. В верхней части рисунка изображен n-уровневый стек приложения, работающий на основе его инфраструктуры. До появления приложений на базе Platform система Facebook работала по совершенно такой же архитектуре. Важнее всего то, что в обеих архитектурах бизнес-логика (включая требования конфиденциальности Facebook) фактически выполняется по правилам, устанавливаемым в некотором компоненте данных системы.

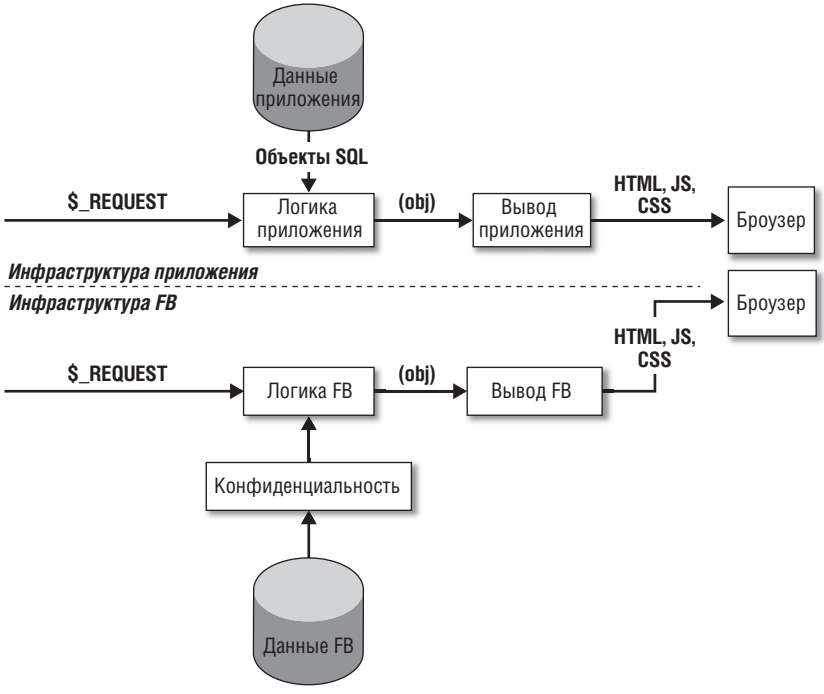


Рис. 6.1. *Раздельные стеки Facebook и n-уровневого приложения*

Рост объема и релевантности данных приводит к тому, что бизнес-логика может предоставлять более персонализированный контент, так что просмотр книг для рецензирования, чтения или приобретения на сайте *http://fettermansbooks.com* (или в любом другом аналогичном приложении) может быть основательно *расширен* за счет социальных данных пользователя из Facebook. А именно, отзывы о книгах, написанные друзьями, списки пожеланий и покупок могут помочь пользователю принять решение, открыть для себя новые книги или укрепить связи с другими пользователями. Если бы внутреннее отображение Facebook *user_get_friends* было доступно для внешних приложений (например, *http://fettermansbooks.com*), это позволило бы включить в данные приложения мощный социальный контекст и избавило бы их от необходимости создавать собственные социальные сети. Интеграция социальных данных принесла бы большую пользу в широком спектре приложений, потому что разработчики могли бы применять базовые отображения Facebook к бесчисленным веб-сайтам, в которых пользователи производят *или* потребляют контент.

Технологии Facebook Platform достигают этой цели при помощи ряда изменений в архитектуре социальных сетей и данных:

- Приложения могут обращаться к полезным социальным данным через службы данных платформы Facebook Platform. Полученный таким образом социальный контекст добавляется во внешние веб-приложения, приложения настольных ОС и приложения для альтернативных устройств.
- Приложения могут публиковать свой вывод на языке разметки FBML. Результаты работы приложения интегрируются со страницами *http://facebook.com*.
- Использование FBML требует определенных изменений в архитектуре. Разработчики могут использовать cookie Facebook Platform и Facebook JavaScript (FBJS), чтобы свести к минимуму изменения, необходимые для публикации присутствия приложения на *http://facebook.com*.
- Наконец, приложения могут пользоваться всеми этими возможностями без ущерба для *конфиденциальности* и ожиданий относительно *восприятия системы пользователем*, сформированных Facebook для пользовательских данных и выводимых результатов.

Последний пункт особенно интересен. Архитектура Facebook Platform не всегда красива – она является первопроходцем во Вселенной социальных платформ. Большая часть архитектурных решений, принимаемых для создания общедоступного социального контекста, формируется под влиянием этого дуализма «инь-ян»: доступности данных и конфиденциальности пользователя.

Создание социальной веб-службы

Даже такой простой пример, как *http://fettermansbooks.com*, очевидно показывает, что многие интернет-приложения выиграли бы от добавления социального контекста в свои данные. Однако мы сталкиваемся с проблемой: доступностью этих данных.

Проблема: использование социальных данных Facebook было бы полезно для приложения, но эти данные недоступны.

Решение: предоставление доступа к данным Facebook через внешнюю веб-службу (рис. 6.2).

Включение Facebook API в архитектуру Facebook начинает формировать связь между внешними приложениями и Facebook через Facebook Platform. В сущности, данные Facebook включаются в стек внешнего приложения. Для пользователя Facebook эта интеграция начинается в тот

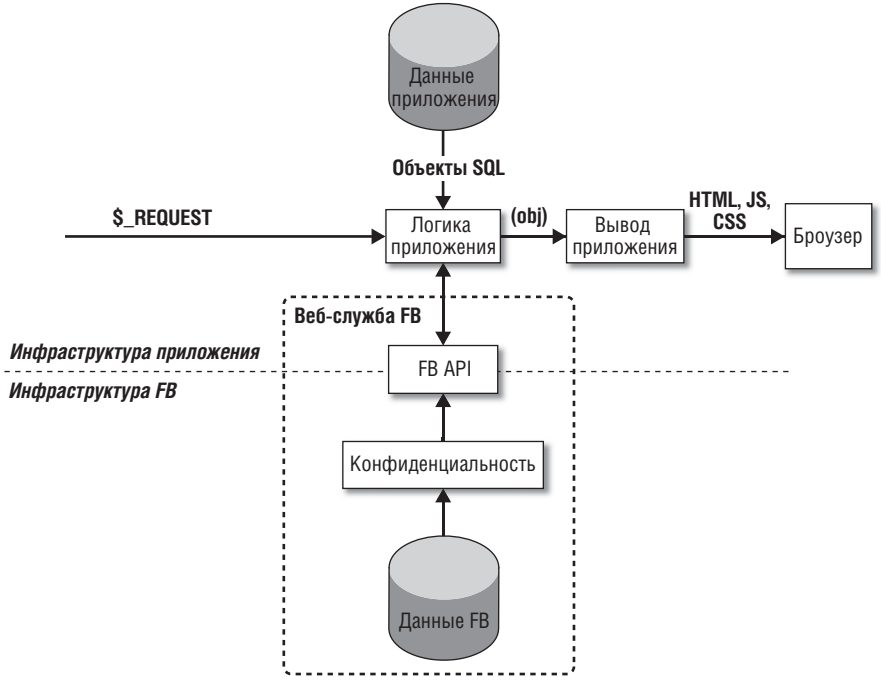


Рис. 6.2. *Стек приложения потребляет данные Facebook в виде веб-службы*

момент, когда он дает явное разрешение на передачу своих социальных данных внешним приложениям.

В листинге 6.4 показан примерный код целевой страницы *http://fettermansbooks.com* без интеграции с Facebook.

Листинг 6.4. Примерная логика книжного сайта

```
$books_user_id = establish_booksite_userid($_REQUEST);
$book_infos = user_get_likely_books($books_user_id);
display_books($book_infos);
```

Функция *user_get_likely_books* работает только с данными, находящимися под контролем приложения (возможно, используя методы анализа релевантности для определения интересов пользователя).

Но представьте, что Facebook предоставляет пользователям внешних сайтов два простых метода RPC (Remote Procedure Call):

- `friends.get()`
- `users.getInfo($users, $fields)`

Располагая этими методами и отображением идентификаторов пользователей *http://fettermansbooks.com* на идентификаторы пользователей Facebookm, мы можем включить социальный контекст в любой контент сайта *http://fettermansbooks.com*. В листинге 6.5 приведена новая реализация логики книжного сайта для пользователей Facebook.

Листинг 6.5. Логика книжного сайта с социальным контекстом

```
$books_user_id = establish_booksite_userid($_REQUEST);
$facebook_client = establish_facebook_session($_REQUEST,$books_user_id);

if ($facebook_client) {
    $facebook_friend_uids = $facebook_client->api_client->friends_get();
    foreach($facebook_friend_uids as $facebook_friend) {
      $book_site_friends[$facebook_friend]
           = books_user_id_from_facebook_id ($facebook_friend);
    }
    $book_site_friend_names = $facebook->api_client->
      users_getInfo($facebook_friend_uids, 'name');

  foreach($book_site_friends as $fb_id => $booksite_id) {
    $friend_books = user_get_reviewed_books($booksite_id);
    print "<hr>" . $book_site_friend_names[$fb_id] . "'s likely picks: <br>";
    display_books($friend_books);
  }
}
```

Жирным шрифтом выделены те места, где внешнее приложение использует данные Facebook Platform. Если бы мы могли получить доступ к коду функции `establish_facebook_session`, то эта архитектура могла бы предоставить доступ к большему количеству данных, и приложение в большей степени ориентировалось бы не только на информацию о книгах, но и на работу с предпочтениями конкретного пользователя.

Давайте посмотрим, как организована поставка социальных данных через Facebook API. Мы начнем с простого технического анализа веб-службы, инкапсулирующей данные Facebook. Служба была создана посредством передачи соответствующих метаданных универсальному генератору кода Thrift. Разработчики могут использовать приемы, описанные в следующем разделе, для эффективного построения веб-служб любого рода независимо от открытости или приватности данных в хранилище разработчика.

Однако следует учитывать, что пользователи Facebook не считают свои данные Facebook абсолютно открытыми. По этой причине после технического обзора мы рассмотрим механизм обеспечения конфиденциальности уровня Facebook посредством главного механизма аутентификации Platform API – пользовательских сеансов.

Данные: создание веб-службы XML

Чтобы включить в приложение простейшую поддержку социального контекста, мы используем вызовы двух удаленных методов, `friends.get` и `users.getInfo`. Внутренние функции, непосредственно работающие с данными, скорее всего, находятся в одной из библиотек кодового дерева Facebook и обслуживают сходные запросы на сайте Facebook. Примеры показаны в листинге 6.6.

Листинг 6.6. Примеры отображений социальных данных

```
function friends_get($session_user) { ... }
function users_getInfo($session_user, $input_users, $input_fields) { ... }
```

Теперь мы переходим к построению простой веб-службы, которая преобразует входные запросы GET и POST через HTTP в вызовы внутреннего стека и выводит результаты в формате XML. В случае Facebook Platform имя вызываемого метода и аргументы передаются в запросе HTTP вместе с регистрационными данными, специфическими для вызывающего приложения («ключ api»), для конкретной пары «пользователь-приложение» («ключ пользовательского сеанса») и для экземпляра запроса («сигнатура» запроса). Мы вернемся к описанию ключа сеанса позднее, в разделе «Аутентификация простой веб-службы»). На верхнем уровне алгоритм обработки запросов к *http://api.facebook.com* выглядит так:

1. Проанализировать полученные регистрационные данные («Аутентификация простой веб-службы»), чтобы проверить подлинность вызывающего приложения, текущую авторизацию пользователя в этом приложении и подлинность запроса.
2. Интерпретировать входящий запрос GET/POST как вызов метода с осмысленными аргументами.
3. Перенаправить вызов внутреннему методу и получить результат в виде структур данных, находящихся в памяти.
4. Преобразовать эти структуры в известный выходной формат (например, XML или JSON), и вернуть управление.

Трудности с конструированием интерфейсов, применяемых внешними пользователями, обычно возникают на этапах 2 и 4. Очень важно обеспечить постоянное сопровождение, синхронизацию и документирование этих интерфейсов данных для внешнего потребителя, однако писать соответствующий код вручную – дело неблагодарное и долгое. Возможно, нам также потребуется открыть доступ к этим данным внутренним службам, написанным на многих языках, или передавать результаты внешнему разработчику по разным веб-протоколам, таким как XML, JSON или SOAP.

Изящное решение проблемы основано на использовании метаданных для инкапсуляции типов и сигнатур, описывающих API. Программисты Facebook создали кросс-языковую систему межпроцессных коммуникаций (IPC, Inter-Process Communication) с открытым кодом Thrift (*http://developers.facebook.com/thrift*), которая помогает элегантно решить эту задачу.

В листинге 6.7 приведен пример файла *.thrift* для нашего API версии 1.0, который преобразуется пакетом Thrift во внутренние механизмы API.

Листинг 6.7. Определение веб-службы для Thrift

```
xsd_namespace http://api.facebook.com/1.0/
/***
* Определение типов api.facebook.com версии 1.0
*/
typedef i32 uid
typedef string uid_list
typedef string field_list

struct location {
 1: string street xsd_optional,
 2: string city,
 3: string state,
 4: string country,
 5: string zip xsd_optional
}

struct user {
 1: uid uid,
 2: string name,
 3: string books,
 4: string pics,
 5: location current_location
}

service FacebookApi10 {

 list<uid> friends_get()
  throws (1:FacebookApiException error_response),

 list<user> users_getInfo(1:uid_list uids, 2:field_list fields)
  throws (1:FacebookApiException error_response),
}
```

Каждый тип в этом примере является примитивным типом (`string`), структурой (`location`, `user`) или обобщенной коллекцией (`list<uid>`). Так как каждое объявление метода обладает типизованной сигнатурой, код, определяющий многократно используемые типы, может быть

сгенерирован на любом языке. В примере 6.8 показана часть вывода, сгенерированного для PHP.

Листинг 6.8. Код службы, сгенерированный Thrift

```
class api10_user {

 public $uid = null;
 public $name = null;
 public $books = null;
 public $pic = null;
 public $current_location = null;

public function __construct($vals=null) {
  if (is_array($vals)) {
   if (isset($vals['uid'])) {
    $this->uid = $vals['uid'];
   }
   if (isset($vals['name'])) {
    $this->name = $vals['name'];
   }
   if (isset($vals['books'])) {
    $this->books = $vals['books'];
   }
   if (isset($vals['pic'])) {
    $this->pic = $vals['pic'];
   }
   if (isset($vals['current_location'])) {
    $this->current_location = $vals['current_location'];
   }
   // ...
 }
 // ...
}
```

Все внутренние методы, возвращающие тип `user` (такие как внутренняя реализация `users_getInfo`), создают все необходимые поля. Их примерный вид показан в листинге 6.9:

Листинг 6.9. Последовательное использование сгенерированного типа

```
return new api_10_user($field_vals);
```

Например, если `current_location` присутствует в объекте `user`, то в какой-то момент перед выполнением листинга 6.9 `$field_vals['current_location']` присваивается значение `new api_10_location(...)`.

Схема выходного кода XML и сопроводительный документ XSD (XML Schema Document) генерируются по именам и типам полей. В листин-

ге 6.10 показан пример реального вывода XML, полученного в результате вызова RPC.

Листинг 6.10. Вывод XML при вызове веб-службы

```
<users_getInfo_response list="true">
 <users type="list">
  <user>
   <name>Dave Fetterman</name>
   <books>Zen and the Art, The Brothers K, Roald Dahl</books>
   <pic></pic>
   <current_location>
    <city>San Francisco</city>
    <state>CA</state>
    <zip>94110</zip>
   </current_location>
  </user>
 </users>
</users_getInfo_response>
```

Thrift генерирует похожий код для объявления вызовов функций RPC, сериализации в известные выходные структуры данных и преобразования внутренних исключений во внешние коды ошибок. Другие инструментарии (такие как XML-RPC и SOAP) также предоставляют некоторые из перечисленных возможностей – вероятно, с повышением нагрузки на процессор и канал связи.

Применение такого изящного инструмента, как Thrift, предоставляет ряд долгосрочных преимуществ:

Автоматическая синхронизация типа

Включение `'favorite_records'` в тип `user` или преобразование `uid` в `i64` должно произойти во всех методах, потребляющих или генерирующих эти типы.

Автоматическое генерирование привязок

Вся хлопотная работа по чтению и записи типов исчезает, а объявления функций, проверки типов и обработка ошибок, необходимые для преобразования вызовов функций в методы RPC, генерирующие код XML, Thrift выполняет автоматически.

Автоматизация документирования

Thrift генерирует общедоступный документ XML Schema Document, который содержит точную информацию для внешних пользователей – обычно гораздо более точную по сравнению с той, которая содержится в «руководствах» и описаниях. Документ также может

использоваться напрямую некоторыми внешними инструментами для генерирования привязок на стороне клиента.

Кросс-языковая синхронизация

Возможно как внешнее использование службы клиентами XML и JSON, так и внутреннее использование через сокет демонами, написанными на любых языках (PHP, Java, C++, Python, Ruby, C# и т. д.). Для этого код должен генерироваться на основе метаданных, чтобы проектировщику службы не приходилось обновлять код клиентов с каждым незначительным изменением.

Итак, у нас появился компонент данных социальной веб-службы. Теперь необходимо разобраться, как назначать ключи сеансов для соблюдения модели конфиденциальности, которую пользователи ожидают от любого расширения Facebook.

Аутентификация простой веб-службы

Простая схема аутентификации обеспечит доступ к социальным данным с учетом представлений о конфиденциальности пользователей Facebook. Пользователь Facebook получает некоторое представление данных системы в зависимости от того, кем он является, от своих настроек конфиденциальности, а также настроек конфиденциальности других пользователей, которые с ним связаны. Пользователи могут разрешить отдельным приложениям унаследовать это представление. Информация, видимая пользователю через внешнее приложение, составляет значительную часть информации, непосредственно видимой пользователю на сайте Facebook (но не более того).

В архитектуре с отдельным внешним приложением (рис. 6.1) аутентификация пользователя часто воплощается в виде передачи cookies от броузера (изначально cookie назначается пользователю после выполнения проверочных действий на сайте). Однако в схеме на рис. 6.2 cookies недоступны для Facebook – внешнее приложение запрашивает информацию от платформы без участия броузера. Для решения этой проблемы в системе Facebook создаются отображения данных пользователя на ключи сеансов, как показано в листинге 6.11.

Листинг 6.11. Отображение данных пользователя на ключ сеанса

```
get_session: {user_id, application_id} -> session_key
```

Клиент веб-службы просто отправляет `session_key` с каждым запросом, чтобы веб-служба знала, от чьего имени выполняется запрос. Если пользователь (или Facebook) отключил это приложение или никогда не использовал его, то проверка безопасности не проходит, и возвра-

щается признак ошибки. В противном случае внешнее приложение использует ключ сеанса в своей собственной системе учета пользователей или в cookie пользователя.

Но как получить этот ключ? Соответствующая логика находится в функции `establish_facebook_session` кода приложения `http://fettermansbooks.com` (см. листинг 6.5). У каждого приложения имеется свой уникальный «ключ приложения» (также называемый `api_key`), с которого начинается последовательность авторизации приложения (рис. 6.3):

1. Пользователь перенаправляется в подсистему входа Facebook с известным `api_key`.
2. Пользователь вводит свои регистрационные данные в Facebook, чтобы авторизовать приложение.
3. Пользователь перенаправляется на целевой сайт проверенного приложения с ключом сеанса и идентификатором пользователя.
4. Теперь приложение может обращаться с вызовами к конечной точке API от имени пользователя (до истечения срока действия или удаления приложения).

Чтобы помочь пользователю начать выполнение этой процедуры, можно вывести специальную ссылку или кнопку:

```
<a href="http://www.facebook.com/login.php?api_key=abc123">
```

с ключом приложения (допустим, `"abc123"`). Если пользователь согласится авторизовать приложение с использованием парольной формы Facebook (естественно, Facebook ни при каких условиях не будет экспортировать пароль), он направляется обратно на сайт приложения с действительным ключом `session_key` и своим идентификатором пользователя Facebook. Ключ сеанса является строго секретным, поэтому для дальнейшей верификации при вызовах передается хеш-код, сгенерированный на основе общего секрета.

Если считать, что разработчик сохранил свои значения `api_key` и секрета приложения[1], код `establish_facebook_session` достаточно просто пишется по схеме на рис. 6.3. Хотя подробности реализации в таких схемах согласования (handshake) могут отличаться, очень важно, чтобы авторизация пользователя становилась возможной только после ввода пароля на Facebook. Интересно заметить, что некоторые ранние приложения просто использовали эту процедуру согласования в качестве собственной парольной системы, вообще не используя данные Facebook.

---

[1] *http://wiki.developers.facebook.com/index.php/Authorization_and_Authentication_for_Desktop_Applications – Примеч. перев.*

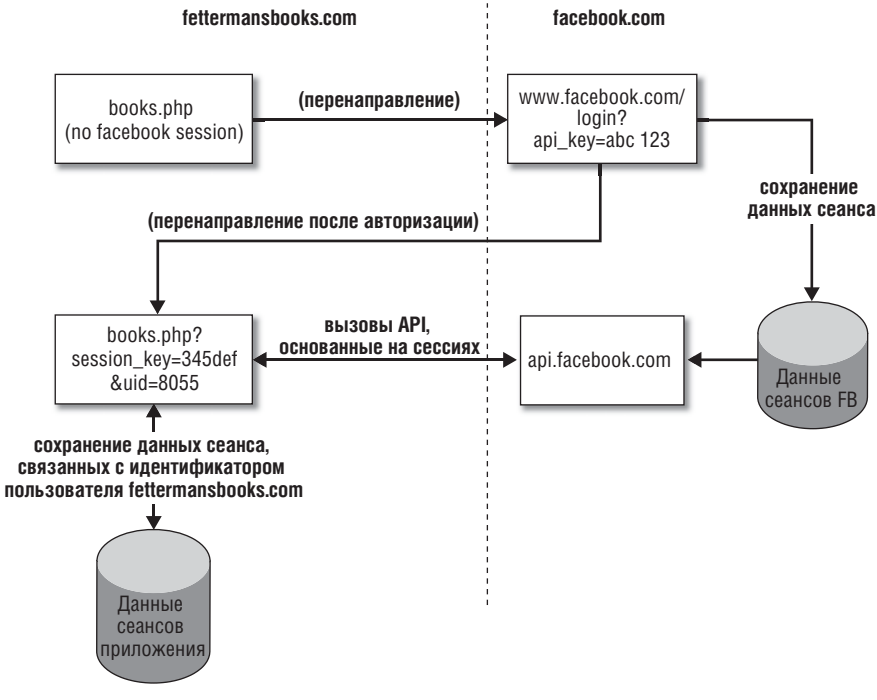


Рис. 6.3. *Авторизация доступа к Facebook Platform API*

Впрочем, некоторые приложения не слишком хорошо приспособлены для второго «перенаправления»: «настольные» приложения, приложения для устройств (например, мобильных телефонов) или встроенные в броузер – однако и эти приложения тоже могут быть весьма полезными. В таком случае применяется несколько иная схема с использованием вторичного маркера (token) авторизации. Маркер запрашивается приложением через API, передается Facebook при первом входе, а затем заменяется приложением на сеансовый ключ и секрет уровня сеанса после аутентификации пользователя на сайте.

Создание социальной службы запросов данных

Итак, мы вывели свои внутренние библиотеки во внешний мир, создав веб-службу с процедурой аутентификации под управлением пользователя. После этого простого изменения социальные данные Facebook могут использоваться в стеке любого другого приложения, авторизован-

ного пользователем. Через социальный контекст, представляющий всеобщий интерес, в данных такого приложения создаются новые отношения.

Как бы гладко ни проходил этот обмен данными в представлениях пользователя, разработчик, использующий платформенные API, знает, что наборы данных весьма различны. Схемы обращения разработчика к его собственным данным сильно отличаются от схем, используемых с данными Facebook. Прежде всего, данные Facebook «живут» на другой стороне запроса HTTP, и вызовы методов через многие соединения HTTP увеличивает задержку и затраты ресурсов в страницах разработчика. Кроме того, база данных внешнего приложения также обеспечивает более высокую детализацию доступа, чем несколько десятков методов Facebook Platform API. Использование собственных данных и знакомого языка запросов (такого как SQL) позволяет разработчику отобрать из таблицы только нужные ему поля, отсортировать или отфильтровать результаты, провести проверку по альтернативным индексам или организовать вложение запросов. Если API платформы не позволяет разработчику выполнять расширенную обработку данных на сервере платформы, разработчику часто приходится импортировать заведомо лишние данные, а затем выполнять стандартные логические преобразования на своих серверах после получения. Это может быть серьезным бременем.

Проблема: получение данных от Facebook Platform API требует значительно больших затрат, чем получение внутренних данных.

С повышением трафика или интенсивности использования приложения, потребляющего данные с внешней платформы, такие факторы, как загрузка канала связи, загрузка процессора и задержка запросов, начинают быстро накапливаться. Несомненно, у проблемы должно существовать хотя бы частичное решение. В конце концов, разве мы не проводили оптимизацию на уровне данных в стеке своего отдельного приложения? Не существует ли технологий, обеспечивающих выборку нескольких наборов данных за один вызов? Нельзя ли выполнить выборку, фильтрацию и сортировку непосредственно на уровне данных?

Решение: реализация внешнего доступа к данным с применением запросов – механизма, используемого при обращении к внутренним данным.

Решение для данных Facebook – FQL – подробно описано позднее, в разделе «FQL». FQL имеет много общего с SQL, но интерпретирует платформенные данные как поля и таблицы (в отличие от свободно определяемых объектов в формате XML). Это позволяет разработчику применять стандартную семантику запросов к данным Facebook – вероятно,

работать с ними так же, как разработчик работает с собственными данными. В то же время преимущества от перемещения обработки на сторону платформы аналогичны преимуществам от перемещения операций уровня данных в SQL. В обоих случаях разработчик сознательно избегает выполнения операций в логике приложения.

FQL представляет улучшенную архитектуру данных, основанную на внутренних данных Facebook, и является следующим шагом после стандартных веб-служб, работающих по принципу «черного ящика». Но сначала необходимо понять, почему поочередная пересылка многих запросов данных неэффективна, и описать простой и очевидный способ ее предотвращения.

Пакетная обработка вызовов

Простейшее решение проблем загрузки родственно методу Facebook API `batch.run`. Чтобы избавиться от задержки, связанной с пересылкой нескольких вызовов *http://api.facebook.com* по стеку HTTP, мы накапливаем входные данные нескольких методов в одном пакете, а затем возвращаем выходные деревья XML в одном ответе. Примерная реализация этой схемы на стороне клиента показана в листинге 6.12.

Листинг 6.12. Пакетное объединение вызовов методов

```
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$notifications = &$facebook->api_client->notifications_get();
$facebook->api_client->end_batch();
```

В клиентской библиотеке PHP5 для Facebook Platform метод `end_batch` инициирует запрос к платформенному серверу, получает все результаты и обновляет ссылочные переменные, используемые для каждого результата. Данные пользователя, относящиеся к одному сеансу, *читаются* в пакетном виде. Обычно механизм пакетной обработки запросов используется для группировки нескольких операций *записи* – например, массовых обновлений профилей Facebook или масштабных оповещений пользователей.

Факт применения пакетной обработки для операций записи не случаен – в нем проявляется главная проблема пакетной обработки. Каждый вызов не должен зависеть от результатов всех остальных вызовов. Операции записи для множества разных пользователей обычно удовлетворяют этому условию, но одна стандартная проблема остается нерешенной: использование результатов одного вызова в качестве входных данных следующего вызова. В листинге 6.13 представлен типичный сценарий, который не подойдет для системы пакетной обработки.

Листинг 6.13. Некорректное использование пакетной обработки

```
$fields = array('uid, 'name', 'books', 'pic', 'current_location');
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$user_info = &$facebook->api_client->users_getInfo($friends, $fields); // НЕТ!
$facebook->api_client->end_batch();
```

Естественно, содержимое `$friends` не существует на момент отправки клиентом запроса `users_getInfo`. Модель FQL элегантно решает эту и другие аналогичные проблемы.

FQL

FQL – простой язык запросов для работы с внутренними данными Facebook. В выходных данных обычно используется тот же формат, что и в Facebook Platform API, но во входных данных простая модель библиотек RPC заменяется моделью запросов, напоминающей SQL: именованные таблицы и поля с установленными отношениями. По аналогии с SQL эта технология предоставляет возможность выборки по экземплярам или диапазонам, выборки подмножества полей из записи данных, вложения запросов для выполнения большего объема работы на сервере данных и устранения необходимости повторных вызовов через стек RPC.

Например, чтобы получить поля с именами `'uid'`, `'name'`, `'books'`, `'pic'` и `'current_location'` для всех пользователей, которые являются моими друзьями, в модели «чистого API», следует выполнить процедуру, приведенную в листинге 6.14.

Листинг 6.14. Цепной вызов методов на стороне клиента

```
$fields = array('uid', 'name', 'books', 'pic', 'current_location');
$friend_uids = $facebook->api_client->friends_get();
$user_infos = users_getInfo($friend_uids, $fields);
```

Такая схема приводит к повышению количества обращений к серверу данных (два вызова), повышению задержки и созданию большего количества точек потенциальных сбоев. Вместо этого для просмотра данных пользователя с идентификатором 8055 (ваш покорный слуга) следует использовать синтаксис FQL с единственным вызовом, как показано в листинге 6.15.

Листинг 6.15. Цепной вызов методов на стороне сервера с использованием FQL

```
$fql = "SELECT uid, name, books, pic, current_location FROM profile
        WHERE uid IN (SELECT uid2 from friends where uid1 = 8055)";
$user_infos = $facebook->api_client->fql_query($fql);
```

На концептуальном уровне данные, на которые ссылается `users_getInfo`, интерпретируются как *таблица* с несколькими *полями*, для которой построен индекс (`uid`). При грамотном использовании эта новая грамматика открывает ряд новых возможностей доступа к данным:

- Интервальные запросы (например, по временным промежуткам)
- Вложенные запросы (`SELECT fields_1 FROM table WHERE field IN (SELECT fields_2 FROM .... )`)
- Фильтрация и сортировка результатов

Архитектура FQL

Разработчики выполняют запросы FQL при помощи функции API `fql_query`. Суть проблемы заключается в объединении «объектов» и «свойств» внешнего API с «таблицами» и «полями» FQL. Мы по-прежнему наследуем последовательность выполнения стандартных вызовов API: выборка данных внутренними методами, применение правил, обычно связываемых с вызовами API, и преобразование вывода в соответствии с системой Thrift (см. ранее раздел «Данные: создание веб-службы XML»). Для каждого метода API, выполняющего чтение данных, в FQL существует соответствующая «таблица», которая абстрагирует данные, стоящие за запросом. Например, метод API `users_getInfo`, который выдает поля `name`, `pic`, `books` и `current_location`, доступные для пользователя с заданным идентификатором, представлен в FQL в виде таблицы `user` с соответствующими полями. Внешний вывод `fql_query` также соответствует выводу стандартной функции API (если документ XSD изменен таким образом, чтобы допустить пропуск полей в объекте), поэтому вывод `fql_query` для таблицы `user` совпадает с выводом соответствующего вызова `users_getInfo`. Более того, такие вызовы, как `user_getInfo`, на стороне сервера Facebook часто реализуются в виде вызовов FQL!

Примечание

На момент написания книги язык FQL поддерживал только команду `SELECT`. Команды `INSERT`, `UPDATE`, `REPLACE`, `DELETE` и т. д. не поддерживались, поэтому на FQL могли быть реализованы только методы чтения. Впрочем, большинство данных, с которыми работают методы Facebook Platform API, все равно доступно только для чтения.

Давайте возьмем в качестве примера таблицу `user` и построим систему FQL для поддержки запросов к ней. Представьте, что под всеми уровнями абстракции данных Facebook Platform (внутренние вызовы, внешний вызов API `users_getInfo`, новая таблица FQL `user`) в базе дан-

ных Facebook действительно присутствует таблица с именем `'user'` (листинг 6.16).

Листинг 6.16. Пример таблицы данных Facebook

```
> describe user;
+--------------+--------------+-----+
| Field        | Type         | Key |
+--------------+--------------+-----+
| uid          | bigint(20)   | PRI |
| name         | varchar(255) |     |
| pic          | varchar(255) |     |
| books        | varchar(255) |     |
| loc_city     | varchar(255) |     |
| loc_state    | varchar(255) |     |
| loc_country  | varchar(255) |     |
| loc_zip      | int(5)       |     |
+--------------+--------------+-----+
```

Допустим, в стеке Facebook для обращения к этой таблице используется следующий метод:

```
function user_get_info($uid)
```

Он возвращает объект на выбранном нами языке (PHP), который обычно используется перед применением логики конфиденциальности и отображением на сайте *http://facebook.com*. Наша реализация веб-службы делала практически то же самое: она преобразовывала контент GET/POST веб-запроса в такой вызов, получала сходный объект, применяла правила конфиденциальности, а затем использовала Thrift для перевода данных в формат XML (рис. 6.2).

«Упаковка» `user_get_info` в FQL позволит нам на программном уровне применить эту модель с объединением таблиц, полей, внутренних функций и правил конфиденциальности в логичную форму, подходящую для повторного использования.

Далее перечислены основные объекты, создаваемые при вызове FQL из листинга 6.15, а также методы, описывающие отношения между ними. Обсуждение подробностей разбора строк, реализации грамматики, построения альтернативных индексов, вычисления пересечений запросов и реализации многих объединяющих выражений (сравнений, принадлежности, конъюнкции и дизъюнкции) выходит за рамки этой главы. Наше внимание будет сосредоточено на том, что относится к данным: высокоуровневое определение соответствующих данным полей и таблиц FQL и преобразование входной команды в запросы к функциям `can_see` и `evaluate` каждого поля (листинг 6.17).

Листинг 6.17. Поля и таблицы FQL

```
class FQLField {
  // Например, table="user", name="current_location"
  public function __construct($user, $app_id, $table, $name) { ... }

  // Отображение: "index" id -> {0,1} (видимо или невидимо)
  public function can_see($id) { ... }
  // Отображение: "index" id -> Thrift-совместимый объект данных
  public function evaluate($id) { ... }
}

class FQLTable {
  // Статический список содержащихся полей:
  // Отображение: () -> ('books' => 'FQLUserBooks', 'pic' ->'FQLUserPic', ...)
  public function get_fields() { ... }
}
```

Новый механизм обращения к данным основан на объектах `FQLField` и `FQLTable`. Объект `FQLField` содержит привязанную к данным логику преобразования индекса «записи» (например, идентификатора пользователя) и информации о просматривающей стороне (`user` и `app_id`) в вызовы данных внутреннего стека. Обязательный метод `can_see` обеспечивает встроенную проверку конфиденциальности. При обработке запроса в памяти создается один объект `FQLTable` для каждой именованной таблицы (`'user'`) и для каждого именованного поля (для `'books'`, для `'pic'` и т. д.). Все объекты `FQLField`, связанные с одним объектом `FQLTable`, обычно используют одну функцию доступа к данным (в следующем примере `user_get_info`), хотя это и не является строго обязательным – всего лишь удобный интерфейс.

В листинге 6.18 приведен пример типичного строкового поля таблицы `user`.

Листинг 6.18. Отображение библиотеки данных на определение поля FQL

```
// Базовый объект для всех простых полей FQL таблицы user.
class FQLStringUserField extends FQLField {

  public function __construct($user, $app_id, $table, $name) { ... }

  public function evaluate($id) {
   // Вызов внутренней функции
   $info = user_get_info($id);
   if ($info && isset($info[$this->name])) {
    return $info[$this->name];
   }
   return null;
  }
```

```
  public function can_see($id) {
   // Вызов внутренней функции
    return can_see($id, $user, $table, $name);
   }
  }

// Простое строковое поле данных
class FQLUserBooks extends FQLStringUserField { }

// Простое строковое поле данных
class FQLUserPic extends FQLStringUserField { }
```

Поля `FQLUserPic` и `FQLUserBooks` различаются только своим внутренним свойством `$this->name`, задаваемым конструктором в ходе выполнения. Обратите внимание: во внутренней реализации для всех операций с данными используется вызов `user_get_info`; такое решение хорошо работает только в том случае, если система кэширует эти результаты в памяти процесса. Реализация Facebook работает именно таким образом, а весь запрос обрабатывается примерно за то же время, что и стандартный вызов платформы API.

Перейдем к более сложному полю, представляющему `current_location`. Объект получает те же входные данные и используется по тем же правилам, но выводит уже встречавшийся нам ранее «структурный» объект (листинг 6.19).

Листинг 6.19. Нетривиальный объект поля данных FQL

```
// Сложный объект поля данных.
class FQLUserCurrentLocation extends FQLStringUserField {
  public function evaluate($id) {
   $info = user_get_info($id);
   if ($info && isset($info['current_location'])) {
    $location = new api10_location($info['current_location']);
   } else {
    $location = new api10_location();
   }
   return $location;
  }
}
```

Такие объекты, как `api10_location`, относятся к сгенерированным типам (см. «Данные: создание веб-службы XML»); Thrift и служба данных Facebook умеют возвращать их в виде XML с правильной структурой. Теперь мы видим, почему даже при новом оформлении входных данных вывод FQL может сохранять совместимость с выводом Facebook API.

Главный цикл обработки `FQLStatement` в листинге 6.21 дает высокоуровневое представление о реализации FQL. В этом коде используются

типы FQLExpression, но для простого запроса в основном речь идет о типах FQLFieldExpression, инкапсулирующих внутренние вызовы в собственные методы evaluate и can_see класса FQLField (листинг 6.20).

Листинг 6.20. Простой класс выражения FQL

```
class FQLFieldExpression {

  // Создается вместе с экземпляром FQLField в свойстве "field"
  public function evaluate($id) {
   if ($this->field->can_see($id))
    return $this->field->evaluate($id);
   else
    return new FQLCantSee(); // преобразуется в сообщение об ошибке
                             // или пропущенное поле
  }

  public function get_name() {
    return $this->field_name;
  }
}
```

Процедура обработки начинается с того, что SQL-подобная строка преобразуется при помощи *lex* и *yacc* в главный массив выражения $select и выражение $where типа FQLStatement. Функция evaluate() типа FQLStatement возвращает запрашиваемые объекты. Главный цикл обработки команды из листинга 6.21 в этом простом высокоуровневом описании выполняет следующие действия:

1. Получить все ограничения для индексов возвращаемых записей. Например, при выборке из таблицы user такими ограничениями будут запрашиваемые идентификаторы пользователей (UID). А если, скажем, запрос обращен к таблице событий, индексированной по времени, ограничения определяют границы временного промежутка.
2. Преобразовать ограничения в канонические идентификаторы таблицы. Запросы к таблице user также могут производиться по полю name; если в выражении FQL используется name, то функция использует внутреннее отображение user_name ->user_id.
3. Для каждого потенциального идентификатора проверить, соответствует ли он «правостороннему» условию (булевская логика, сравнения, операции IN и т. д.). Если не соответствует, исключить его из результата.
4. Обработать каждое выражение (в нашем случае поля секции SELECT) и создать элемент XML в форме *<COL_NAME>COL_VALUE</COL_NAME>*, где *COL_NAME* – имя поля FQLTable, а *COL_VALUE* – результат обработки поля функцией evaluate соответствующего объекта FQLField.

Листинг 6.21. Основной цикл обработки FQL

```
class FQLStatement {

 // Члены класса:
 // $select: массив объектов FQLExpression из секции SELECT запроса,
 //  соответствующих, например, "books", "pic" и "name".
 // $from: объект FQLTable для исходной таблицы
 // $where: объект FQLExpression с ограничениями для запроса.
 // $user, $app_id: значения user и app_id вызывающей стороны.

 public function __construct($select, $from, $where, $user, $app_id) { ... }

  // Список всех известных таблиц в системе FQL.
 public static $tables = array(
  'user'        => 'FQLUserTable',
  'friend'      => 'FQLFriendTable',
  );

// Возвращение элементов XML, преобразуемых в выходные данные
public function evaluate() {

  // На основании содержимого секции WHERE мы сначала
  // получаем набор выражений, представляющих ограничения
  // индексируемых столбцов, упоминаемых в секции WHERE

  // Получить все "правосторонние" константы для идентификаторов
  // (например, X в выражении 'uid = X')
  $queries = $this->where->get_queries();

  // Преобразовать в индекс записи. Если 'name' используется
  // в качестве альтернативного индекса таблицы user, то здесь
  // значение преобразуется в uid.
  $index_ids = $this->from_table->get_ids_for_queries($queries);

  // Отфильтровать набор по содержимому WHERE и параметрам LIMIT
  $result_ids = array();

  foreach ($ids as $id) {
     $where_result = $this->where->evaluate($id);

        // Проверить, соответствует ли запись ограничениям 'WHERE',
        // если она не ограничена правилами конфиденциальности
        if ($where_result && !($where_result instanceof FQLCantSee))
            $result_ids [ ]= $id;
  }

  $result = array();
  $row_name = $this->from_table->get_name(); // например, "user"

  // Заполнить массив результата запрашиваемыми данными
  foreach ($result_ids as $id) {
    foreach ($this->select as $str => $expression) {
                               // Например, "books" или"pic"
```

```
      $name = $expression->get_name();
      $col = $expression->evaluate($id); // Возвращает значение
      if ($col instanceof FQLCantSee)
        $col = null;

      $row->value[] = new xml_element($name, $col);
    }

    $result[] = $row;
    }
    return $result;
  }
```

У FQL есть немало других тонкостей, но эта общая процедура показывает, как на основе внутреннего доступа к данным в сочетании с реализацией политики конфиденциальности создается совершенно новая

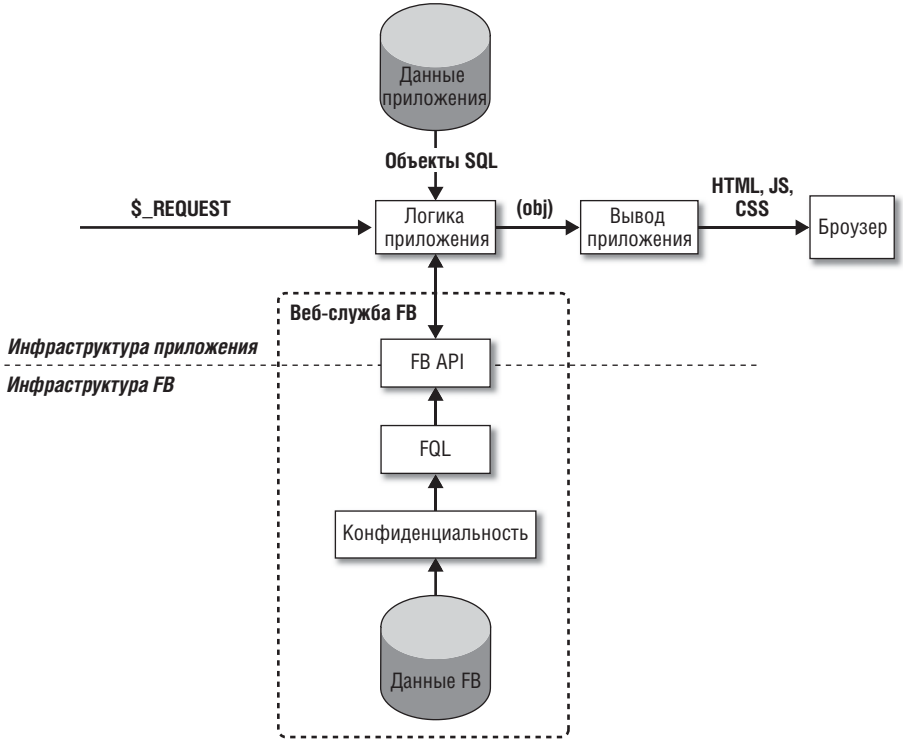


Рис. 6.4. *Стек приложения потребляет данные Facebook через веб-службу и запросы*

модель запросов. Это позволяет разработчику ускорить обработку запросов и обращаться к данным с более высокой степенью детализации, чем при использовании API, сохраняя знакомый синтаксис SQL.

По мере того как внутренняя реализация многих функций API переходит на инкапсуляцию соответствующих методов FQL, общая архитектура развивается до состояния, показанного на рис. 6.4.

Создание социального веб–портала: FBML

Сервис Facebook позволяет стекам внешних приложений интегрировать данные социальных платформ в свои системы; это весьма полезная возможность. Эти архитектуры данных повышают доступность данных социальной платформы: пользователи, работающие как с внешним приложением (например, *http://fettermansbooks.com*), так и с платформой данных (например, *http://facebook.com*), могут обмениваться социальной информацией между собой, что снимает необходимость в создании новой социальной сети для каждого нового социального приложения. Но даже с этими возможностями такие приложения еще не обладают полной мощью таких социальных платформ, как Facebook. Чтобы приложение приносило реальную пользу, оно должно стать известным многим пользователям. В то же время не все внутренние данные, поддерживающие социальную платформу, можно сделать доступными для внешних стеков. Сейчас мы последовательно разберем каждую из этих проблем, которые должен решить создатель платформы.

Проблема: чтобы социальные приложения достигли «критической массы», при которой они начинают привлекать новых посетителей, пользователи поддерживающей социальной ветви должны знать о взаимодействиях других пользователей с этими приложениями. Это предполагает более глубокую интеграцию приложения с социальным сайтом. Проблема с популяризацией данных, продукта или системы существовала с первых дней разработки программных продуктов. Отсутствие пользователей создает особенно большие трудности в пространстве Web 2.0 – если пользователи не будут потреблять и (особенно) генерировать контент, то какую пользу принесут наши системы?

Facebook объединяет множество пользователей, заинтересованных в обмене информацией по социальным каналам, и может отображать не только собственный контент, но и контент внешних приложений. Присутствие внешнего приложения на сайте Facebook ускорит популяризацию приложений, написанных крупными и малыми разработчиками, и поможет им набрать количество пользователей, необходимое для обеспечения хорошей социальной функциональности.

Внешнее приложение, каким бы оно ни было, должно быть четко и очевидно представлено на сайте Facebook. Facebook Platform представляет нашим приложениям такую возможность; URL-путь вида *http://apps.facebook.com/fettermansbooks/...* резервируется для контента приложения, выводимого на сайте Facebook. Вскоре мы покажем, как платформа интегрирует данные, логику и вывод приложения.

Вторая проблема является естественным следствием специфики наших служб данных, построенных в разделах «Данные: создание веб-службы XML» и «FQL», и решить ее ничуть не проще.

Проблема: внешние приложения не могут использовать некоторые базовые элементы данных, доступ к которым Facebook не предоставляет через свои веб-службы.

Facebook предоставляет значительный объем данных в распоряжение пользователей, публикующих свой контент непосредственно на сайте (*http://facebook.com*), но не вся эта информация предоставляется внешними службами данных. Хорошим примером могут служить сами данные конфиденциальности (отображение `can_see` из раздела «Основные данные Facebook») – они не видны напрямую пользователям сайта Facebook, а отображение `can_see` остается невидимым для внешних служб данных. Тем не менее соблюдение настроек конфиденциальности, заданных пользователем на Facebook, является признаком хорошо интегрированного приложения, учитывающего пожелания пользователей социальной системы. Как же разработчикам использовать данные, которые Facebook для сохранения конфиденциальности пользователей не публикует через свои службы данных?

Самое элегантное решение проблемы основано на встраивании данных Facebook в данные, логику и вывод внешнего приложения, которые при этом работают в среде, пользующейся доверием пользователя.

Решение: разработчики создают прикладной контент для вывода и выполнения на самом сайте социальной сети. Для создания приложения используется управляемый данными язык разметки, интерпретируемый Facebook.

Приложения, использующие только элементы Facebook Platform из разделов «Данные: создание веб-службы XML» и «FQL», создают социальные взаимодействия, внешние по отношению к Facebook и расширенные за счет использования сервиса социальных данных Facebook. В архитектуре данных и веб-архитектуре, описанных в этом разделе, сами приложения превращаются в своего рода службу данных, поставляющую контент Facebook для отображения на сайте *http://apps.facebook.com*. URL вида *http://apps.facebook.com/fettermansbooks/...* уже не соответствует данным, логике и выводу Facebook, а обращается к служ-

бе по адресу *http://fettermansbooks.com* для генерирования контента приложения.

При обсуждении этой модели нам следует одновременно учитывать как активы, так и пассивы. С одной стороны, мы получаем социальную систему с высоким трафиком, на которой пользователи узнают о внешнем контенте, и доступ к большому объему социальных данных, дополняющих социальные приложения. С другой стороны, запросы должны исходить с социального сайта (Facebook), использовать приложение как службу данных и выводить контент в формате HTML, JavaScript и CSS – и все это без нарушения конфиденциальности или ожиданий пользователей Facebook.

Начнем с рассмотрения нескольких *неправильных* решений этой задачи.

Приложения на Facebook: прямая генерация HTML, CSS и JS

Допустим, в конфигурации внешнего приложения появились два новых поля с именами `application_name` и `callback_url`. Если они содержат имя `"fettermansbooks"` и URL *http://fettermansbooks.com/fbapp/* соответственно, то это означает, что сайт *http://fettermanbooks.com* будет обслуживать запросы пользователей к URL вида *http://apps.facebook.com/fettermansbooks/PATH?QUERY_STRING* на своих собственных серверах, по адресу *http://fettermansbooks.com/fbapp/PATH?QUERY_STRING*.

Запрос к *http://apps.facebook.com/fettermansbooks/...* просто получает контент HTML, JS и CSS на серверах приложений и отображает его в качестве главного контента страницы на Facebook. Фактически Facebook работает с внешним сайтом как с *веб-службой HTML*.

Такое решение существенно изменяет n-уровневую модель приложения. Ранее стек, потребляющий данные Facebook, обслуживал запросы, обращенные непосредственно к *http://fettermansbooks.com*. Теперь под корневым веб-узлом приложения располагается дерево, которое само предоставляет данные HTML. Facebook получает контент из запросов к этой новой службе приложения (которая, в свою очередь, может потреблять данные Facebook), снабжает его в стандартными элементами навигации сайта Facebook и отображает для пользователя.

Но если Facebook будет отображать код HTML, JavaScript и CSS приложения прямо на своих страницах, это позволит приложению полностью нарушить ожидания пользователя относительно более контролируемой среды *http://facebook.com*, а сам сайт и всего его пользователи станут уязвимы для сетевых атак. Разрешать внешним пользователям

прямую модификацию разметки и сценариев почти всегда нежелательно. Собственно, внедрение кода или сценариев обычно является целью атакующих, поэтому польза от такой «возможности» выглядит сомнительной.

Вдобавок эта схема не открывает доступа к дополнительным данным, не экспортируемым API. Хотя такая схема закладывает фундамент изменений в стеке приложения, она не обеспечивает полноценного решения ни одной из имеющихся проблем.

Приложения на Facebook: iframe

Очевидный подход к более безопасному отображению контента одного приложения в визуальном и логическом контексте другого сайта базируется на технологии, уже встроенной непосредственно в броузер: `iframe`.

Для конфигурации из предыдущего раздела при запросе к *http://apps.facebook.com/fettermansbooks/PATH?QUERY_STRING* будет сгенерирован код HTML следующего вида:

```
<iframe src="http://fettermansbooks.com/fbapp/PATH?GET_STRING"></iframe>
```

Контент по данному URL будет отображаться во фрейме на странице Facebook, а в его собственной автономной среде может использоваться любой тип веб-технологий: HTML, JS, AJAX, Flash и другие.

По сути броузер начинает выполнять функции брокера запросов вместо прямого представления контента Facebook. В отличие от модели из предыдущего раздела, броузер также обеспечивает безопасность остальных элементов сгенерированной страницы, поэтому разработчики могут запрограммировать содержимое фрейма так, как считают нужным.

Для приложений, разработчики которых желают с минимальными усилиями переместить свой код с логики сайта на Platform, технология `iframe` вполне оправдана. Более того, Facebook продолжает поддерживать модель `iframe` для генерирования полных страниц. Хотя данное решение позволяет добиться первой цели – интеграции внешнего контента в социальный сайт, вторая цель остается под вопросом. Даже с безопасностью потока запросов на базе `iframe` эти разработчики не смогут пользоваться новыми данными, не экспортируемыми API.

Приложения на Facebook: язык разметки FBML

Каждое из наивных решений, представленных в двух предыдущих разделах, обладает своей прелестью. Решение на базе HTML идет по интуитивно понятному пути переоформления приложений в виде веб-

служб и возврата контента для отображения на сайте Facebook. Модель `iframe` обеспечивает выполнение внешнего контента в изолированной (и безопасной) среде. Оптимальное решение сочетает удобство модели «внешнего приложения как службы» и безопасности `iframe`, позволяя разработчикам использовать больший объем социальных данных.

Проблема заключается в том, что для реализации своего неповторимого социального приложения разработчики должны предоставить данные, логику и вывод из стека своего приложения. Однако вывод должен генерироваться с пользовательскими данными, которые не должны выходить за пределы Facebook.

Как решить проблему? Возвращать не код HTML, а особый язык разметки, в котором определяется значительная часть логики и внешнего вида приложения, а также запросы к защищенным данным, и поручить Facebook сгенерировать код вывода в защищенной серверной среде! Эта идея послужила отправной точкой для создания FBML (рис. 6.5).

В этой схеме запрос к *http://apps.facebook.com* снова преобразуется в запрос к приложению, и стек приложения снова использует службу данных Facebook. Однако вместо того чтобы возвращать код HTML, разработчик переписывает приложение так, чтобы оно возвращало код FBML. Этот код содержит многие элементы HTML, но дополняется специальными тегами, определяемыми Facebook. Получив данные по запросу, интерпретатор FBML на сайте Facebook в ходе построения страницы приложения преобразует полученную разметку в собственные данные, исполняемые элементы и элементы вывода. Пользователь получает страницу, которая состоит из обычных веб-элементов страниц Facebook, но включает интегрированные данные, логику и элементы оформления приложения. Какой бы код ни возвращался приложением, FBML позволяет Facebook выполнить ожидания пользователей относительно конфиденциальности и качества пользовательских взаимодействий на технологическом уровне.

FBML представляет собой особую разновидность XML. Многие теги разметки знакомы нам по HTML, но к ним добавились платформенно-зависимые теги для отображения на Facebook. В FBML используется тот же высокоуровневый паттерн, что и в FQL: известный стандарт (HTML, или SQL в случае FQL) изменяется для отложенного выполнения и принятия решений на сервере Facebook Platform. Как видно из рис. 6.5, интерпретатор FBML позволяет самому разработчику управлять логикой выполнения и выводом на сервере Facebook на уровне *данных* FBML. Это отличный пример модели выполнения, в центре которой находятся данные: FBML работает на *декларативном* уровне, а не на *императивном* (в отличие от решений на C, PHP и т. д.).

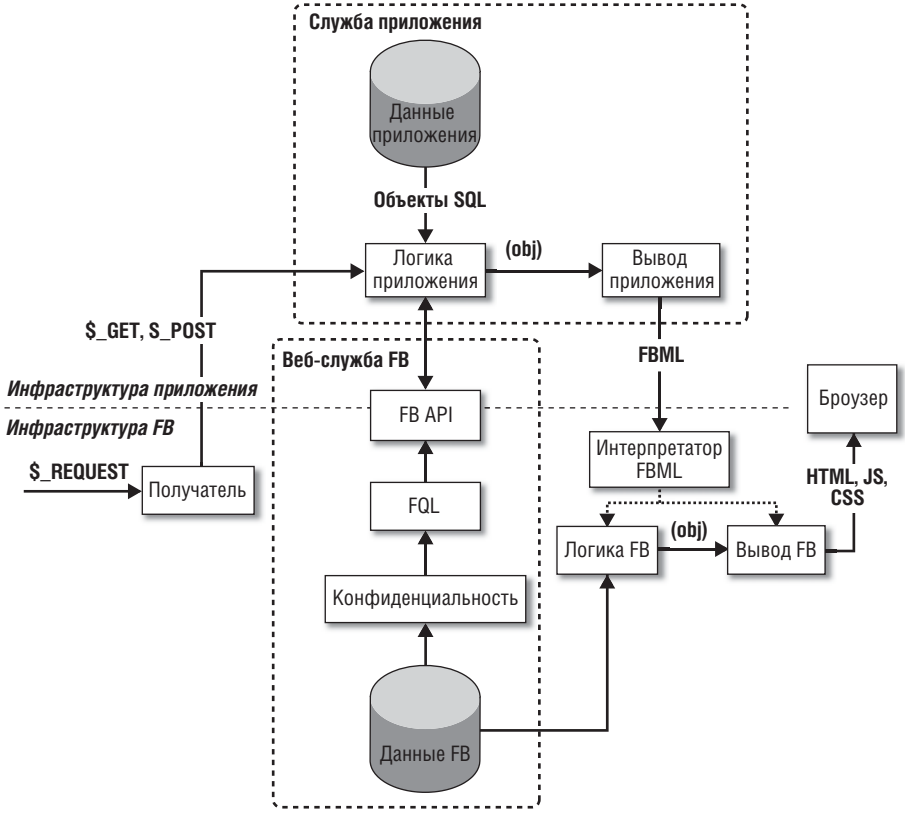


Рис. 6.5. *Приложения как службы FBML*

Пора заняться подробностями. FCML является диалектом XML, поэтому синтаксис языка состоит из тегов, атрибутов и контента. Теги делятся на несколько широких концептуальных категорий.

Теги HTML

Если служба FBML возвращает тег `<p/>`, то Facebook выводит его в выходную страницу как `<p/>`. В области вывода поддерживается большинство тегов HTML, за исключением немногих, нарушающих политику конфиденциальности или дизайна уровня Facebook.

Итак, строка FBML `<h2>Hello, welcome to <i>Fetterman's books!</i></h2>` останется неизменной при выводе в HTML.

Теги вывода данных

В этой категории проявляется мощь данных Facebook. Представьте, что профильные картинки не передаются за пределы сайта. Указав тег `<fb:profile-pic uid="8055">`, разработчик может использовать больший объем данных пользователя Facebook в своем приложении, причем пользователю не нужно полностью доверять эту информацию разработчику.

Например, тег:

```
<fb:profile-pic uid="8055" linked="true" />
```

преобразуется в следующую разметку:

```
<a href="http://www.facebook.com/profile.php?id=8055"
 onclick="(new Image()).src = '/ajax/ct.php?app_id=...">
  <img uid="8055" linked="true" src="..." alt="Dave Fetterman"
                  title="Dave Fetterman" />
</a>
```

Примечание

Сложный атрибут `onclick` генерируется для того, чтобы ограничить применение кода JavaScript при выводе изображения на странице Facebook.

Обратите внимание: даже если информация защищена, контент не возвращается в стек приложения, а только выводится для просмотра пользователем. Выполнение на стороне контейнера позволяет просматривать эти данные, не передавая их через приложение!

Теги обработки данных

Следующий пример еще лучше демонстрирует использование скрытых данных. Пользовательские ограничения конфиденциальности, доступные только через внутренний метод `can_see` (листинг 6.3), являются важной частью взаимодействия с приложением; тем не менее, они не экспортируются через службу данных Facebook. При помощи тега `<fb:-if-can-see>` и других аналогичных тегов приложение, например, может указать в атрибутах целевого пользователя, чтобы дочерние элементы выводились только в том случае, если «зрителю» разрешен просмотр контента целевого пользователя. Таким образом, параметры конфиденциальности не передаются приложению, однако приложение может использовать их для обеспечения конфиденциальности.

В этом смысле FBML является доверенной декларативной средой выполнения, в отличие от императивных сред выполнения типа C или PHP. Формально FBML не является «полным по Тьюрингу», в отличие

от этих языков (например, в нем нет циклических конструкций). Как и в языке HTML, во время выполнения невозможно хранить состояние (кроме того, которое подразумевается обходом вершин дерева; например, элемент <fb:tab-item> имеет смысл только во <fb:tabs>). И все же FBML предоставляет большую часть той функциональности, которую многие разработчики хотели бы реализовать для своих пользователей, через доступ к данным в безопасной системе.

FBML фактически позволяет *определить* логику и вывод выполняемого приложения, но при этом уникальный контент приложения начинается с серверов приложений.

Теги дизайна

Facebook часто хвалят за дизайн, и многие разработчики пытаются сохранить «облик и поведение» Facebook, повторяя элементы дизайна. Часто они копируют код JavaScript и CSS с сайта *http://facebook.com*, но FBML приносит с собой нечто вроде библиотеки «макросов дизайна», которые позволяют добиться той же цели с меньшим риском.

Например, Facebook применяет известные классы CSS, которые преобразуют ввод типа <fb:tabs>...</fb:tabs> в конкретную структуру вкладок в верхней части страницы разработчика. Эти элементы дизайна также могут содержать семантику выполнения; например, теги <fb:narrow>...</fb:narrow> выводят контент своих дочерних элементов в FBML только в том случае, если результат выполнения выводится в узком столбце пользовательского профиля.

В листинге 6.22 приведен код FBML с тегами дизайна.

Листинг 6.22. Пример кода FBML

```
<fb:tabs>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/mybooks.php"
 title='My Books' selected='true'/>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/recent.php"
 title='Recent Reviews' />
</fb:tabs>
```

Приведенный фрагмент преобразуется в набор визуальных вкладок, ведущих к определенному контенту. В ходе преобразования используются пакеты Facebook для работы с HTML, CSS и JavaScript.

Теги замены HTML

HTML не создает сколько-нибудь заметного риска для безопасности и конфиденциальности данных, так что теги замены HTML в FBML предназначены разве что для изменения или ограничения некоторых

наборов параметров (например, автовоспроизведения Flash). Их присутствие не является строго обязательным для вывода; просто эти теги обеспечивают соответствие приложения стандартному поведению сайта-контейнера. Впрочем, такие изменения начинают играть важную роль по мере того, как экосистема приложений развивается по направлению к имитации внешнего вида и поведения контейнера.

Рассмотрим следующий пример кода FBML:

```
<fb:flv src=http://fettermansbooks.com/newtitles.flv height="400"
  width="400" title="New Releases">
```

Этот фрагмент преобразуется в довольно длинную строку JavaScript для отображения модуля воспроизведения видео; данный элемент находится под управлением Facebook и намеренно запрещает некоторые аспекты поведения (например, автовоспроизведение).

Теги «пакетной функциональности»

Некоторые теги Facebook FBML включают целые пакеты стандартной функциональности приложений Facebook. Тег `<fb:friend-selector>` создает пакет для выбора друзей с упреждающим вводом с клавиатуры; эта функция, присутствующая на многих страницах Facebook, включает данные Facebook (друзья, основные сети), стили CSS и код JavaScript для обработки нажатий клавиш. Такие теги поощряют применение стандартных шаблонов дизайна и общих элементов приложений в приложениях, а также ускоряют реализацию соответствующего поведения разработчиками.

FBML: небольшой пример

Вспомните, как отразилось на нашем гипотетическом внешнем сайте введение функций API `friends.get` и `users.getInfo` в исходный код `http://fettermansbooks.com`. Теперь мы рассмотрим пример того, как FBML объединяет социальные данные с бизнес-логикой конфиденциальности, создавая впечатление полностью интегрированного приложения.

Если бы нам удалось получить обзоры книг, используя обращение к базе данных вида `book_get_all_reviews($isbn)`, то мы могли бы объединить данные друзей, настройки конфиденциальности и Стену для вывода обзоров на Стене. Соответствующий код FBML на сайте-контейнере приведен в листинге 6.23.

Листинг 6.23. Создание приложения на базе FBML

```
// Социальные обзоры книг на Стене Facebook
// Использованы теги FBML: <fb:profile-pic>, <fb:name>,
```

```
// <fb:if-can-see>, <fb:wall>>

// Из раздела 1.3
$facebook_friend_uids = $facebook_client->api_client->friends_get();
  foreach($facebook_friend_uids as $facebook_friend) {
    if ($books_user_id = books_user_id_from_facebook_id($facebook_friend))
      $book_site_friends[] = $books_user_id;
   }
}

// Гипотетическое отображение, возвращающее
// объект books_uid -> book_review
$all_reviewers = get_all_book_reviews($isbn);

$friend_reviewers = array_intersect($book_site_friends,
                    array_keys($all_reviewers));

echo 'Friends' reviews:<br/>';
echo '<fb:wall>';

// Разместить друзей в начале списка.
foreach ($friend_reviewers as $book_uid => $review) {
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
  echo '</fb:wallpost>';
unset($all_reviewers[$book_uid]); // Не включать недрузей.
}

echo 'Other reviews:<br/>';

// Остались только недрузья.
foreach ($all_reviewers as $book_uid => $review) {
  echo '<fb:if-can-see uid="'.$book_uid.'">'; // По умолчанию поиск
                                              // выполняется только
                                              // среди доступных
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
  echo '</fb:wallpost>';
  echo '</fb:if-can-see>';
}

echo '</fb:wall>';
```

Хотя на этот раз вместо веб-вызова с выводом HTML служба выдает код FBML, обычная последовательность выполнения остается неизменной. Данные Facebook позволяют приложению вывести более релевантные обзоры (написанные друзьями) до менее релевантных, а код FBML используется для вывода результата с использованием соответствующей логики конфиденциальности и элементов дизайна Facebook.

Архитектура FBML

Преобразование кода FBML, предоставленного разработчиками, в код HTML, отображаемый на сайте *http://facebook.com*, требует совместного применения нескольких технологий и концепций: разбора входной строки в дерево синтаксиса, интерпретации тегов дерева как вызовов внутренних методов, применения правил синтаксиса FBML и соблюдения ограничений сайта-контейнера. Как и в случае с FQL, наше внимание сосредоточено прежде всего на взаимодействии FBML с платформенными данными, а остальные фрагменты технологической головоломки только упоминаются в самых общих чертах. FBML решает сложную задачу, и подробное описание реализации FBML заняло очень много места – нам пришлось бы рассказать о таких пропущенных темах, как регистрация ошибок FBML, возможность предварительного кэширования контента для последующего вывода, цифровая подпись результатов отправки форм в целях безопасности, и т. д.

Начнем с низкоуровневой проблемы разбора FBML. Платформа Facebook, наследующая некоторые роли броузера, также наследует и некоторые из его проблем. Для удобства разработчиков мы не требуем, чтобы входные данные проверялись по схеме или даже представляли собой XML в правильном формате – незакрытые теги HTML (скажем, `<p>` вместо соответствующего стандарту XHTML тега `<p/>`) нарушают предположение о том, что входные данные могут быть разобраны как правильный код XML. По этой причине нам необходимо сначала преобразовать входной поток FBML в правильно сформированное дерево синтаксиса с тегами, атрибутами и контентом.

Для решения этой задачи был использован открытый код из кодовой базы броузера. В этой главе данная фаза процесса не рассматривается; будем считать, что после получения кода FBML и его отправки у нас появляется древовидная структура с именем `FBMLNode`, которая предоставляет возможность запрашивать теги, пары атрибутов «ключ-значение» и низкоуровневый контент для любого узла в сгенерированном дереве синтаксиса, а также запрашивать дочерние элементы в рекурсивном режиме.

Переходя на более высокий уровень, обратите внимание, что код FBML повсеместно встречается на сайте Facebook: на «холстах» приложений, в контенте поставок новостей, в контенте профильных полей и в ряде других мест. Каждый из этих контекстов FBML определяет ограничения на входные данные; например, страницы холстов допускают использование `iframe`, а профильные поля нет. И это естественно, поскольку FBML поддерживает конфиденциальность данных практически на таком же уровне, как в API, контекст выполнения должен вклю-

чать как информацию о просматривающем пользователе, так и идентификатор приложения, генерирующего контент.

Итак, прежде чем заняться полезными свойствами FBML, мы начнем с правил контекста, представленных классом `FBMLFlavor` (листинг 6.24).

Листинг 6.24. Класс FBMLFlavor

```
abstract class FBMLFlavor {

// Конструктору передается массив
// с идентификаторами пользователя и приложения
public function FBMLFlavor ($environment_array) { ... }
public function check($category)
 {
  $method_name = 'allows_' . $category;
    if (method_exists($this,$method_name)) {
      $category_allowed = $this->$method_name();
    } else {
      $category_allowed = $this->_default();
    }
    if (!$category_allowed))
      throw new FBMLException('Forbidden tag category '.$category.'
                          in this flavor.');
    }
protected abstract function _default();
}
```

Программа создает экземпляр класса, производного от абстрактного класса контекста, который соответствует странице или элементу, генерирующему FBML. Пример приведен в листинге 6.25.

Листинг 6.25. Конкретизация абстрактного класса FBMLFlavor

```
class ProfileBoxFBMLFlavor extends FBMLFlavor {
protected function _default() { return true; }
public function allows_redirect() { return false; }
public function allows_iframes() { return false; }
 public allows_visible_to() { return $this->_default(); }
 // ...
}
```

Структура класса очень проста: он содержит контекст конфиденциальности (пользователь и приложение) и реализует метод `check`, определяющий правила применения содержательной логики из приведенного позднее класса `FBMLImplementation`. По аналогии с уровнем реализации Platform API класс реализации содержит фактическую логику и данные службы, а остальная часть кода предоставляет доступ к этим методам. У каждого специфического тега Facebook, например

<fb:*TAG-NAME*>, существует соответствующий метод реализации fb_*TAG_NAME* (скажем, метод класса fb_profile_pic реализует логику тега <fb:profile-pic>). У каждого стандартного тега HTML также имеется соответствующий обработчик с именем tag_*TAG_NAME*. Эти обработчики HTML часто пропускают данные в неизменном виде, но в некоторых случаях FBML приходится выполнять проверки и преобразования даже с «обычными» элементами HTML.

Мы рассмотрим реализацию некоторых тегов, а затем соединим все вместе. Каждый метод реализации получает объект FBMLNode, возвращаемый синтаксическим анализатором FBML, и возвращает код HTML в виде строки. Ниже приводятся примеры реализации некоторых тегов HTML, тегов вывода и обработки данных. Учтите, что в листингах используются некоторые функции, которые в тексте подробно не описаны.

Реализация тегов HTML в FBML

В листинге 6.26 приведена внутренняя реализация тега <img> в FBML. Реализация содержит дополнительную логику; иногда URL-адрес исходного изображения заменяется URL-адресом изображения, кэшированного на серверах Facebook. В этом проявляется мощь FBML: стек приложения может возвращать разметку, очень близкую к коду HTML самого сайта, а Facebook обеспечивает необходимое для Platform поведение чисто техническими средствами.

Листинг 6.26. Реализация тега fb:img

```
class FBMLImplementation {
public function __construct($flavor) {... }

 // <img>: пример тега HTML (раздел 4.3.1)
 public function tag_img($node) {

  // В некоторых контекстах FBML изображения запрещены -
  // например, в заголовках поставок
  $this->_flavor->check('images');

  // Удалить пары "ключ-значение" атрибута transform
  // в соответствии с правилами в FBML
  $safe_attrs = $this->_html_rewriter->node_get_safe_attrs($node);
  if (isset($safe_attrs['src'])) {
   // Здесь источник может заменяться изображением
   // из сети доставки контента Facebook
   $safe_attrs['src'] = $this->safe_image_url($safe_attrs['src']);
  }
  return $this->_html_rewriter->render_html_singleton_tag($node->
  get_tag_name(), $safe_attrs);
 }
}
```

Реализация тегов вывода данных в FBML

В листинге 6.27 приведены примеры использования данных Facebook средствами FBML. Тег `<fb:profile-pic>` получает атрибуты `uid`, `size` и `title` и объединяет их в выходном коде HTML на основании внутренних данных и в соответствии со стандартом Facebook. В данном случае выходные данные представляют собой профильную картинку с именем пользователя, связанную с профильной страницей этого пользователя, причем изображение показывается только в том случае, если контент доступен для просматривающего пользователя. Эта функция также находится в классе `FBMLImplementation`.

Листинг 6.27. Реализация тега fb:profile-pic

```
// <fb:profile-pic>: пример тега отображения времени
public function fb_profile_pic($node)
{
  // Если изображения запрещены, значит, профильная картинка
  // тоже запрещена
  $this->check('images');

  $viewing_user = $this->get_env('user');
  $uid = $node->attr_int('uid', 0, true);
  if (!is_user_id($uid))
  throw new FBMLRenderException('Invalid uid for fb:profile_pic ('.$uid .')');

  $size = $node->attr('size', "thumb");
  $size = $this->validate_image_size($size);

  if (can_see($viewing_user, $uid, 'user', 'pic')) {
   // Инкапсулирует функцию user_get_info, которая получает
   // поле данных 'pic' пользователя
   $img_src = get_profile_image_src($uid, $size);
  } else {
   return '';
  }
  $attrs['src'] = $img_src;
  if (!isset($attrs['title'])) {
   // Также можно включить информацию об имени пользователя.
   // Функция также инкапсулирует внутреннюю
   // функцию user_get_info
   $attrs['title'] = id_get_name($id);
  }

  return $this->_html_renderer->render_html_singleton_tag('img', $attrs);
}
```

Теги обработки данных в FBML

Рекурсивная природа разбора FBML делает возможным тег `<fb:if-can-see>` – один из примеров управления *выполнением* средствами FBML, аналог команды `if` в стандартных императивных языках. Реализация тега в классе `FBMLImplementation` представлена в листинге 6.28.

Листинг 6.28. Реализация тега fb:if-can-see

```
// <fb:if-can-see>: пример тега обработки данных
public function fb_if_can_see($node)
{
global $legal_what_values; // Допустимые значения (профиль,
                           // друзья, стена и т.д.)
  $uid = $node->attr_int('uid', 0, true);
  $what = $node->attr_raw('what', 'search'); // По умолчанию используется
                                             // видимость 'search'
if (!isset($legal_what_values[$what]))
return ''; // Неизвестное значение не может быть видимым
  $viewer = $this->get_env('user');
  $predicate = can_see($viewer, $uid, 'user', $what);
  return $this->render_if($node, $predicate); // Обрабатывает else за нас
 }

// Вспомогательная функция для семейства fb_if
 protected function render_if($node, $predicate)
 {
  if ($predicate) {
   return $this->render_children($node);
  } else {
   return $this->render_else($node);
  }
 }

protected function render_else($node)
 {
  $html = '';
  foreach ($node->get_children() as $child) {
   if ($child->get_tag_name() == 'fb:else') {
    $html .= $child->render_children($this);
   }
  }

  return $html;
 }

public function fb_else($ignored_node) { return ''; }
```

Если заданная пара «зритель-объект» проходит проверку `can_see`, ядро FBML выводит дочерние элементы узла `<fb:if-can-see>` в рекурсивном

режиме. В противном случае выводится контент, находящийся под необязательными дочерними элементами <fb:else>. Обратите внимание на то, как fb_if_can_see напрямую обращается к дочерним элементам <fb:else>; if <fb:else> находится перед одним из «if-подобных» тегов FBML, тег и его дочерние теги вообще не возвращают никакого контента. Таким образом, FBML не ограничивается простой заменой; он учитывает структуру документа, а, следовательно, может включать элементы условной логики.

Все вместе

Каждая из описанных функций должна быть зарегистрирована как функция обратного вызова (callback), используемая в ходе разбора входного кода FBML. На Facebook (и в реализации Platform с открытым кодом) этот синтаксический анализатор написан на C как расширение PHP, и каждая из функций обратного вызова находится непосредственно в дереве PHP. Чтобы завершить высокоуровневую логику, мы должны объявить эти теги для синтаксического анализатора FBML. Листинг 6.29, как и другие листинги, основательно отредактирован для упрощения кода.

Листинг 6.29. Основная логика обработки FBML

```
// Входные данные:
// $fbml_impl – реализация, созданная выше
// $fbml_from_callback – исходная строка с кодом FBML,
//                       сгенерированная внешним приложением

// Список "тегов HTML"
$html_special = $fbml_impl->get_special_html_tags();

 // Список специфических тегов FBML (<fb:FOO>)
 $fbml_tags = $fbml_impl->get_all_fb_tag_names();

 // Заменяемые атрибуты всех тегов
 $rewrite_attrs = array('onfocus', 'onclick', /* ... */);

 // Определение групп тегов, передаваемых контекстной функции check()
 // ((например, 'images', 'bold', 'flash', 'forms' и т. д.)
 $fbml_schema = schema_get_schema();

 // Передать ограничения и имена методов обратного вызова
 // внутреннему синтаксическому анализатору FBML, написанному на C.
 fbml_complex_expand_tag_list_11($fbml_tags, $fbml_attrs,
   $html_special,$rewrite_attrs, $fbml_schema);

 $parse_tree = fbml_parse_opaque_11($fbml_from_callback);
 $fbml_tree = new FBMLNode($parse_tree['root']);

 $html = $fbml_tree->render_html($fbml_impl);
```

FBML расширяет броузерную технологию разбора при помощи функций обратного вызова, инкапсулирующих данные, логику выполнения и макросы вывода, создаваемых и находящихся под управлением Facebook. Эта простая идея обеспечивает полноценную интеграцию приложений, а также открывает возможность сознательного экспортирования данных через API с сохранением безопасности работы пользователя. FBML, почти являющийся языком программирования, развивает концепцию данных и выводит ее на новый уровень: внешняя декларативная логика, обеспечивающая безопасное управление данными, выполнение и вывод на Facebook.

Поддержка функциональности системы

Итак, на этой стадии созданный внешним разработчиком программный продукт работает на Facebook; он интегрирован не на уровне отдельных виджетов, а на уровне целых приложений. Попутно мы создали совершенно новую концепцию социального веб-приложения. Все начиналось со стандартной конфигурации с изолированными данными, логикой и выводом типичного веб-приложения, лишенной каких-либо социальных данных кроме тех, которые пользователи согласятся ввести. Мы прошли полный путь к приложению, использующему службы социальных данных Facebook и при этом поставляющему данные FBML для полноценной интеграции в сайте-контейнере.

Данные Facebook прошли долгий путь развития от внутренних библиотек, упоминавшихся в первой части этой главы. Несмотря на это, некоторые важные типичные веб-сценарии и технологии до сих пор не поддерживаются Platform. Преобразовав свое приложение в службу данных, возвращающую FBML (вместо конечной точки HTML/CSS/JS, используемой непосредственно броузером), мы вступаем в конфликт с некоторыми важными допущениями относительно современных веб-приложений. Давайте посмотрим, как Facebook Platform исправляет некоторые из возникших проблем.

Платформенные cookies

Новая веб-архитектура приложений отсекает некоторые технологии, встроенные в броузер, от которых зависит работа многих веб-стеков. Пожалуй, важнее всего то, что броузерные cookies, используемые для хранения информации о взаимодействиях пользователя со стеком приложения, становятся недоступными, поскольку пользователем конечной точки приложения является не броузер, а Facebook Platform.

На первый взгляд может показаться, что проблема удачно решается передачей cookies броузером вместе с запросом к стеку приложения. Од-

нако в этом случае областью действия cookies становится *http://facebook.com*, тогда как на самом деле информация cookies относится к взаимодействиям, предоставляемым областью действия приложения.

Что делать? Поручить Facebook роль броузера. Для этого функциональность cookie дублируется в собственных хранилищах Facebook. Если служба FBML приложения отправляет заголовки, пытающиеся задать броузерные cookies, Facebook просто сохраняет информацию cookie, ассоциированную с парой (пользователь, идентификатор приложения). Затем Facebook «воссоздает» эти cookies, как бы это сделал броузер, при отправке последующих запросов к стеку приложения для данного пользователя.

Такое решение просто реализуется и почти не изменяет ожиданий разработчика при переключении стека HTML на роль службы FBML. Обратите внимание: информация cookie не может использоваться, если пользователь переключится на стек HTML, который также может предоставляться приложением. С другой стороны, взаимодействие пользователя с Facebook, возможно, полезно отделить от взаимодействия с HTML-сайтом приложения.

FBJS

Если стек приложения используется как служба FBML (в отличие от прямого использования броузером пользователя), у Facebook не будет возможности выполнить сценарий на стороне броузера. Прямое возвращение контента в неизменном виде (неудовлетворительное решение, как упоминалось в начале раздела с описанием FBML) могло бы разрешить эту проблему, но оно противоречит устанавливаемым Facebook ограничениям на вывод. Например, Facebook не хочет, чтобы события `onload` открывали всплывающие окна при загрузке профильной страницы пользователя. С другой стороны, полный запрет JavaScript блокирует слишком большую часть полезной функциональности (например, Ajax или динамическое изменение контента страницы без перезагрузки).

Вместо этого FBML интерпретирует содержимое предоставленных деревьев `<script>` и других элементов страниц с учетом описанных ограничений. Вдобавок Facebook предоставляет библиотеки JavaScript для простой, но контролируемой реализации стандартных сценариев. Совместно эти изменения образуют подсистему эмуляции JavaScript для Facebook Platform, сокращенно FBJS, которая позволяет создавать динамические, но при этом безопасные приложения. Для этого используются следующие меры:

- Замена атрибутов FBML для ограничения области действия виртуального документа.
- Выполнение активного сценарного контента откладывается до того момента, как пользователь выполнит действие со страницей или элементом.
- Facebook предоставляет библиотеки для контролируемой реализации типичных сценариев.

Разумеется, эти изменения необходимы не всем сайтам-контейнерам, реализующим собственные платформы. И все же FBJS показывает, какие решения иногда приходится применять для обхода ограничений новых веб-архитектур типа рассмотренной выше. Решения представляются только на уровне общих идей; большая часть FBJS сопряжена с последовательными усовершенствованиями FBML и построением обширных специализированных библиотек JavaScript.

Прежде всего, код JavaScript обычно обладает доступом ко всему дереву DOM (Document Object Model) документа, в котором он содержится. Однако на странице холста Facebook размещает собственные элементы, которые не разрешается изменять внешнему разработчику. Что делать? Снабдить предоставляемые разработчиком элементы HTML и символические имена JavaScript префиксом с идентификатором приложения (например, `app1234567`). В этом случае попытка вызова запрещенной функции `alert()` в коде JavaScript приведет к вызову неопределенной функции `app1234567_alert`, а для таких вызовов JavaScript, как `document.getElementById`, будут доступны только те части кода HTML документа, которые предоставлены самим разработчиком.

Чтобы дать представление о преобразованиях, которым FBJS подвергает предоставленный код FBML (включая элементы `<script>`), мы создадим простую страницу FBML, реализующую функциональность AJAX (листинг 6.30).

Листинг 6.30. Страница FBML с использованием FBJS

```
These links demonstrate the Ajax object:
<br /><a href="#" onclick="do_ajax(Ajax.RAW); return false;">AJAX
Time!</a><br />
<div>
<span id="ajax1"></span>
</div>

<script>
function do_ajax(type) {
var ajax = new Ajax();      // Библиотека FBJS Ajax
 ajax.responseType = type;
 switch (type) {
```

```
<!-- Объект FBJS Ajax также реализует типы AJAX.JSON и AJAX.FBML, которые
здесь не приводятся для экономии места -->
  case Ajax.RAW: ajax.ondone = function(data) {
document.getElementById('ajax1').setTextValue(data);
};
break;
 };

ajax.post('http://www.fettermansbooks.com/testajax.php?t='+type);

}
</script>
```

FBML с изменениями FBJS преобразует этот ввод в код HTML, представленный в листинге 6.31. Комментариями NOTE в этом примере отмечаются все необходимые преобразования; эти комментарии не являются частью выходного кода.

Листинг 6.31. Пример вывода HTML с JavaScript

```
<!-- NOTE 1-->
<script type="text/javascript" src="http://static.ak.fbcdn.net/
.../js/fbml.js"></script>

<!-- Код HTML приложения -->
These links demonstrate the Ajax object:
<br>
<!-- NOTE 2 -->
<a href="#" onclick="fbjs_sandbox.instances.a1234567.bootstrap();
 return fbjs_dom.eventHandler.call(
[fbjs_dom.get_instance(this,1234567),function(a1234567_event) {
a1234567_do_ajax(a1234567_Ajax.RAW);
return false;
}
,1234567],new fbjs_event(event));return true">
AJAX Time!</a>
<br>

<div>

<span id="app1234567_ajax1" fbcontext="b7f9b437d9f7"></span><!-- NOTE 3 -->
</div>

<!-- Код инициализации FBJS, сгенерированный Facebook -->
<script type="text/javascript">
var app=new fbjs_sandbox(1234567);
app.validation_vars={ <!-- Опущено для ясности -->};
app.context='b7f9b437d9f7';
app.contextd=<!-- Опущено для ясности -->;
app.data={"user":8055,"installed":false,"loggedin":true};
app.bootstrap();
```

```
</script>

<!-- Сценарный код приложения -->

<script type="text/javascript">
function a1234567_do_ajax(a1234567_type) { <!-- NOTE 3 -->

var a1234567_ajax = new a1234567_Ajax();<!-- NOTE 3 -->
 a1234567_ajax.responseType = a1234567_type;
 switch (a1234567_type) {
  case a1234567_Ajax.RAW:
a1234567_ajax.ondone = function(a1234567_data) {
a1234567_document.getElementById('ajax1').setTextValue(a1234567_data);
};
break;
};

<!-- NOTE 4 -->
a1234567_ajax.post('http://www.fettermansbooks.com/
testajax.php?t='+a1234567_type);
}
</script>
```

Ниже поясняются комментарии `NOTE` в коде:

NOTE 1

Для вывода сценарного кода разработчик Facebook должен включить свой специализированный код JavaScript, в том числе определение `fbjs_sandbox`.

NOTE 2

Помните элемент `$rewrite_attrs` из приводившегося ранее кода инициализации FBML? FBML заменяет атрибуты в списке функциональностью, специфической для Facebook; в действительности это составная часть FBJS. Таким образом, `onclick` активизирует другие элементы страницы, которые остаются неактивными до выполнения действия пользователем.

NOTE 3

Обратите внимание на префиксы с идентификатором приложения в HTML и сценарном коде. Они означают, что вызов `alert()` будет преобразован в вызов `app1234567_alert()`. Если служебный код JavaScript сайта Facebook разрешает выполнение этого метода в текущем контексте, то управление в конечном итоге будет передано `alert()`, а если нет – вызов будет неопределенным.

Кроме того, префиксы фактически определяют пространства имен в дереве DOM, так что изменения в частях документа ограничиваются частями, определяемыми разработчиком. Другие аналогич-

ные приемы изоляции позволяют разработчикам также применять таблицы CSS (с ограниченной областью действия).

NOTE 4

Facebook предоставляет специализированные объекты JavaScript (такие как `Ajax` и `Dialog`), предназначенные для реализации (и часто усовершенствования) стандартных сценариев. Например, запросы через объект `Ajax()` могут возвращать результат с кодом FBML, поэтому они пересылаются через посредника в Facebook, где Facebook выполняет преобразование FBML в HTML.

Использование FBJS требует изменений в FBML, специализированного кода JavaScript и серверных элементов (скажем, посредника AJAX) для обхода ограничений веб-архитектуры приложения, но результаты впечатляют. Разработчики могут использовать многие возможности JavaScript (и даже некоторые расширенные возможности, такие как поддержка AJAX в FBML). При этом Platform следит за тем, чтобы контент приложения обеспечивал контролируемое поведение, которого пользователи ожидают от Facebook, чисто техническими средствами.

Сводка усовершенствований

В процессе решения оставшихся проблем, возникших из-за введения новой концепции социальной n-уровневой архитектуры, мы снова усовершенствовали архитектуру своей службы данных – см. новые блоки COOKIE и FBJS на рис. 6.6.

В процессе превращения социальных приложений из внешнего сайта, используемого броузером, в интегрированную службу, используемую Facebook, нам пришлось переработать или создать заново часть функциональности броузера (посредством платформенных cookie, FBJS и т. д.). Это всего лишь два примера значительных модификаций, необходимых при попытке изменения или переосмысления идеи «приложения». Facebook Platform содержит ряд дополнительных архитектурных модификаций того же направления, которые не были описаны в этой главе (например, Data Store API и клиент веб-службы на стороне броузера).

Итоги

Социальная информация, предоставляемая пользователями Facebook, определяет полезность фактически любой страницы *http://facebook.com*. Однако эти данные настолько универсальны, что самые полезные применения встречаются при их интеграции со стеками приложений, написанных внешними разработчиками. Интеграция становится

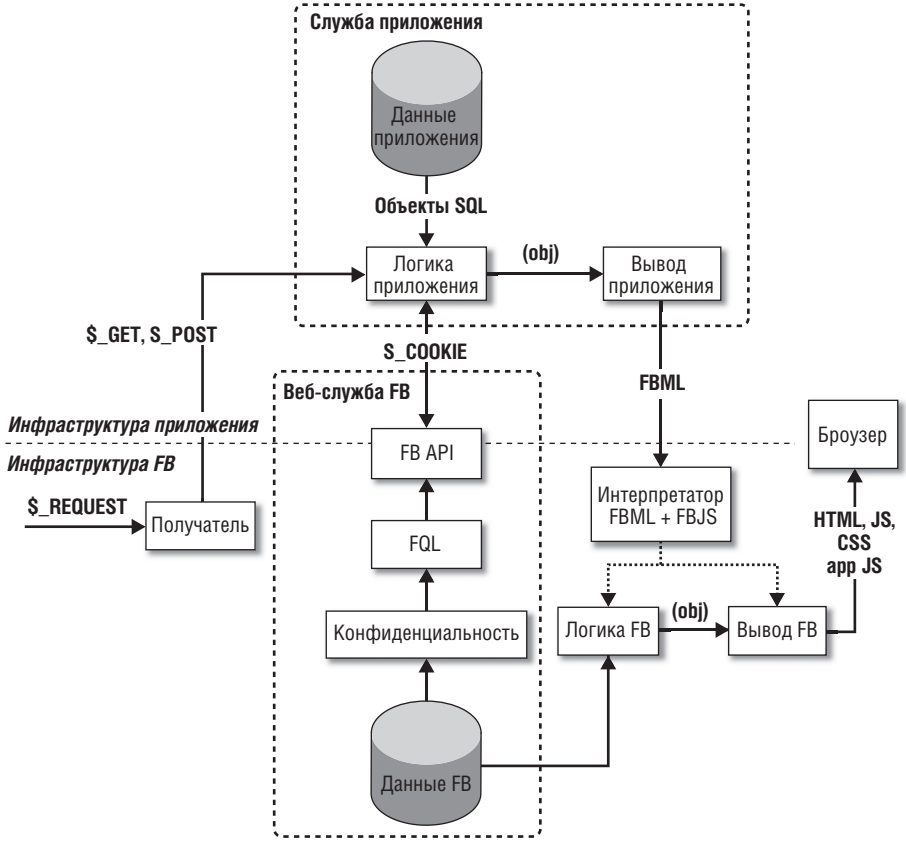


Рис. 6.6. *Архитектура на базе Facebook Platform*

возможной благодаря технологиям данных, таким как веб-службы Facebook Platform, службы получения данных и FBML.

Все усовершенствования, описанные в этой главе, начиная с простого внутреннего API для получения информации о друзьях или профиле пользователя, показывают, как согласовать постоянно развивающиеся методы доступа к данным с ожиданиями сайта-контейнера, прежде всего с требованиями конфиденциальности данных и целостности взаимодействий с сайтом. Каждое новое изменение в архитектуре данных создает новые проблемы в веб-архитектуре, и для решения последних приходится привлекать еще более значительные усовершенствования в схемах обращения к данным.

Хотя наше внимание было сосредоточено исключительно на потенциальных возможностях и ограничениях приложений, построенных с использованием платформы социальных данных Facebook, новые службы данных такого рода не ограничиваются социальной информацией. По мере того как пользователи создают и потребляют все больше информации, которая может представлять интерес для других сайтов-контейнеров (коллекции данных, обзоры, географические сведения, личное планирование, совместная работа и т. д.), применение идей, заложенных в основу уникальной архитектуры данных и веб-архитектуры Facebook Platform, принесет пользу создателям платформ любых видов.

III

Системная архитектура

	Принципы и свойства		Структуры
✓	Гибкость	✓	Модуль
	Концептуальная целостность	✓	Зависимость
✓	Возможность независимого изменения		Обработка
	Автоматическое распространение		Доступ к данным
	Удобство построения		
✓	Адаптация к росту		
	Сопротивление энтропии		

7

Xen и красота виртуализации

Дерек Мюррей
Кайр Фрейзер

Введение

Платформа виртуализации Xen выросла из чисто научного исследования в крупный проект с открытым исходным кодом. Xen позволяет своим пользователям запускать несколько операционных систем на одном физическом компьютере; при этом особое внимание уделяется быстродействию, изоляции и безопасности.

Проект Xen оказал заметное влияние в нескольких областях: от программ до оборудования, от университетской науки до коммерческих разработок. Его успех в значительной мере обусловлен распространением проекта с открытым кодом на условиях лицензии GPL (GNU General Public License). Впрочем, не стоит полагать, что разработчики ни с того, ни с сего решили написать гипервизор с открытым кодом. Существование Xen начиналось в контексте более крупного (и еще более амбициозного) исследовательского проекта под названием *Xenoservers*. Этот проект стимулировал разработку Xen, поэтому мы будем использовать его здесь для объяснения необходимости виртуализации.

Выбор модели с открытым кодом не только обеспечил доступность Xen для широкого круга пользователей, но и позволил использовать преимущества симбиотической связи с другими проектами, также распространяемыми с открытым кодом. Уникальность Xen заключается в том, что в первой версии для запуска таких массовых операционных систем, как Linux, применялся механизм *паравиртуализации*. Паравиртуализация подразумевает внесение изменений в операционные системы, работающие на базе Xen; это приводит как к ускорению их работы, так и к упрощению кода Xen.

С другой стороны, возможности паравиртуализации не безграничны, и запуск немодифицированных операционных систем (таких как Microsoft Windows) возможен только с аппаратной поддержкой со стороны производителей процессоров. Одним из перспективных направлений разработки процессоров является добавление новых функций поддержки виртуальных машин и устранения проблем с быстродействием.

Архитектура Xen медленно развивается с появлением новых функций и нового оборудования. Тем не менее, базовая структура практически не изменилась от первого прототипа до текущей версии. В этой главе мы проследим за тем, как проходило становление архитектуры Xen от первых дней исследовательского проекта, с выпуском трех основных версий, и до сегодняшнего дня.

Xenoservers

Работа над Xen началась в Кембриджском университете в апреле 2002 года. Изначально проект Xen разрабатывался как часть проекта Xenoservers, направленного на создание «глобальной инфраструктуры распределенной обработки».

Примерно в то же время *распределенные вычисления* (*grid computing*)[1] активно продвигались как лучший механизм использования вычислительных ресурсов, распределенных по всему миру. В исходной концепции распределенных вычислений машинное время рассматривается как ресурс (по аналогии с электричеством), который может выде-

[1] *Распределенные вычисления* (*grid computing*) – концепция объединения массива небольших компьютеров в единый виртуальный суперкомпьютер. Сети распределенных вычислений часто объединяют большое количество географически удаленных машин, которые полностью поступают в распоряжение сети или предоставляют другим часть своих вычислительных ресурсов, когда те становятся доступны. Grid-сети получили сегодня широкое распространение в качестве недорогой альтернативы суперкомпьютерам при реализации научных и исследовательских проектов. – *Примеч. ред.*

ляться сетью («решеткой») взаимодействующих компьютеров. Тем не менее в последующих реализациях на первый план вышли *виртуальные организации* – группы компаний и учреждений, между которыми устанавливались доверительные отношения (порой довольно сложные) с применением мощных алгоритмов шифрования с открытым ключом для выполнения аутентификации и авторизации.

Проект Xenoservers подошел к решению задачи с другой стороны. Вместо того чтобы устанавливать доверительные отношения с поставщиками, клиент выбирает ресурс на открытом рынке через брокера *XenoCorp*. Последний хранит список *ксеносерверов* (компьютеров, предлагаемых в аренду третьими сторонами) и распределяет клиентов по серверам, собирая и передавая оплату за их использование. Принципиально здесь то, что между клиентом и поставщиком существуют отношения взаимного *недоверия*: клиент не может повредить компьютеру поставщика, а поставщик не может вмешаться в выполнение задания клиента.

На помощь приходит виртуализация. Вместо того чтобы создавать для клиента учетную запись на сервере, поставщик предоставляет ему отдельную виртуальную машину, которую тот может использовать по своему усмотрению. Клиент запускает на этой машине любую операционную систему и любые приложения (рис. 7.1). Программное обеспечение виртуализации следит за тем, чтобы виртуальная машина была изолирована от остальных частей компьютера (которые могут сдаваться в аренду другим клиентам). *Гипервизор*, под управлением которого работают виртуальные машины, состоит из двух частей: *контрольного монитора*, который следит за тем, чтобы виртуальные машины не могли обращаться к ресурсам других виртуальных машин (особенно данным), и *планировщика*, выделяющего каждой виртуальной машине достойную долю процессорного времени.

Недоверие как полезная архитектурная функция?

На первый взгляд выглядит нелогично. Однако главная цель безопасности в данном случае заключается в том, чтобы другие не могли получить доступ или изменить ваши конфиденциальные данные. Таким образом, доверенная система – это система, которой разрешен доступ к вашим данным. Когда недоверие интегрируется в архитектуру, количество доверенных компонентов сводится к минимуму, а это способствует обеспечению безопасности по умолчанию.

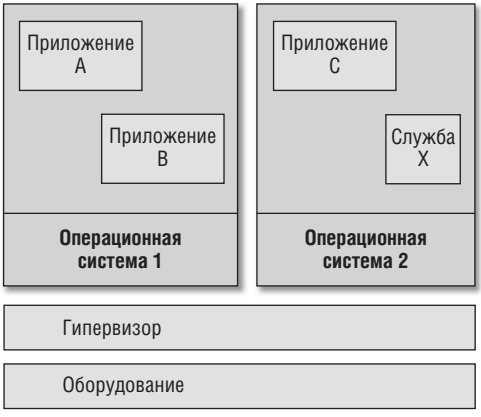


Рис. 7.1. *Архитектура виртуальных машин*

На рис. 7.1 изображены две виртуальные машины, работающие под управлением гипервизора. На первой виртуальной машине работает Операционная система 1 (например, Microsoft Windows) и два приложения; на второй виртуальной машине работает Операционная система 2 (скажем, Linux), приложение и служба.

Оказывается, виртуализация полезна и для других вещей. Например, во многих центрах обработки данных каждому приложению (базе данных, веб-серверу и т. д.) назначается выделенный сервер, но при этом используется лишь незначительная часть ресурсов каждого сервера – процессора, памяти и пропускной способности канала. Естественно, было бы лучше консолидировать все приложения на меньшем количестве физических машин, чтобы сэкономить вычислительные мощности, дисковое пространство и потребности в сопровождении компонентов. Однако простой запуск приложений в одной операционной системе приводит к неудовлетворительным результатам. Параллельное выполнение разных приложений может привести к непрогнозируемому падению производительности. Что еще хуже, возникает риск «цепных реакций», когда сбой одного приложения становится причиной сбоев других приложений. Если же выделить каждому приложению виртуальную машину, а затем запустить эти машины под управлением гипервизора, последний сможет защитить приложения и проследит за тем, чтобы каждому приложению досталась законная часть ресурсов сервера.

Идея применения виртуализации для ресурсных вычислений (utility computing) получила распространение за последние коды. Один из самых известных сервисов ресурсных вычислений – Amazon EC2 – позво-

ляет клиентам создавать виртуальные машины, работающие на серверах в центрах обработки данных Amazon. Клиент оплачивает процессорное время и пропускную способность каналов, используемых его виртуальной машиной. В качестве программного обеспечения виртуализации на этих серверах работает Xen, что приближает сервис EC2 к концепции Xenoservers (хотя в данном случае поддерживается только один поставщик).

Виртуализация также влияет на распределенные вычисления. Globus, стандарт *de facto* программного обеспечения промежуточного уровня

Почему бы не использовать операционную систему?

Операционные системы с разделением времени существуют еще с начала 1960-х годов. Они позволяют одновременно запускать процессы нескольким пользователям с отношениями взаимного недоверия. Так почему бы не предоставить каждому пользователю учетную запись, скажем, на компьютере с системой Unix?

Конечно, такое решение обеспечит совместное использование вычислительных ресурсов. Однако удовлетворительным его не назовешь, потому что пользователи обладают значительно меньшей гибкостью и изоляцией ресурсов.

Что касается гибкости, пользователь может запускать только программы, совместимые с установленной операционной системой; он не сможет запустить другую или изменить текущую операционную систему. В самом деле, пользователю (не обладающему административными привилегиями) не удастся установить программы, требующие разрешений *root*.

Что касается изоляции ресурсов, то ядру операционной системы, которое представляет собой исключительно сложный блок программного кода, будет трудно отслеживать все ресурсы, задействованные конкретным пользователем. В качестве примера можно привести ситуацию «*fork-бомбы*», когда пользователь запускает экспоненциально растущее количество процессов. Запущенные процессы быстро поглощают все ресурсы процессора и парализуют обслуживание других пользователей.

Таким образом, чтобы подобные атаки не проходили, в многопользовательских системах обычно должны присутствовать определенные уровни доверия и этикета между пользователями.

для реализации распределенных вычислений, теперь поддерживает *виртуальные рабочие пространства*, сочетающие виртуальные машины с существующими протоколами безопасности и управления ресурсами в структурах распределенных вычислений. Дополнительное преимущество заключается в том, что виртуальное рабочее пространство – как и любая виртуальная машина – при изменении условий может быть переведено на другую физическую площадку.

Принципиальное преимущество виртуальных машин в модели Xenoservers заключается в том, что они могут использоваться для запуска операционных систем массового использования и существующих приложений. На практике это означает выполнение программ на доминирующей архитектуре x86, вследствие чего разработчик гипервизора сталкивается с рядом проблем.

Проблемы виртуализации

На верхнем уровне абстракции виртуализация операционных систем используется для объединения нескольких виртуальных машин на одной физической машине. Операционные системы запускаются на виртуальных машинах; операционные системы могут запускаться и на физических машинах. Так чем же виртуальная машина отличается от физической?

Самое очевидное различие – это, конечно, оборудование. На физической машине операционная система полностью контролирует все подключенные устройства: сетевые карты, жесткие диски, видеокарты, мышь и клавиатуру. Виртуальные машины не обладают прямым доступом к оборудованию, поскольку это нарушило бы принцип изоляции виртуальных машин. Например, виртуальная машина (сокращенно ВМ) не хочет, чтобы другие ВМ могли обращаться к ее внешней памяти или читать ее сетевые пакеты. Более того, в такой схеме было бы трудно предотвратить злоупотребления. Можно выделить каждой виртуальной машине по одному устройству каждого типа, но это сведет на нет всю экономию от виртуализации. Проблема решается выделением каждой виртуальной машине набора *виртуальных устройств*, предоставляющих ту же функциональность, что и реальные устройства, но мультиплексируемых на физических устройствах.

Более тонкие различия проявляются при запуске операционной системы на виртуальной машине. Традиционно ядро операционной системы является самым привилегированным программным кодом, выполняемым на компьютере; ему разрешается выполнение некоторых команд, недоступных для пользовательских программ. При виртуализации максимальными привилегиями обладает гипервизор, а ядра операци-

онных систем работают на более низком уровне привилегий. Если операционная система попытается выполнить эти команды, у нее ничего не выйдет, но здесь очень важен конкретный механизм сбоя. Если происходит ошибка, перехватываемая гипервизором, то он сможет правильно эмулировать команду и вернет управление виртуальной машине. Однако в архитектуре x86 некоторые команды на низких уровнях привилегий ведут себя иначе – например, «молча» переносят сбой без возможности его перехвата гипервизором. Для виртуализации это крайне неприятно, потому что такой механизм сбоев не позволит операционной системе правильно работать на виртуальных машинах. Разумеется, такие команды необходимо заменить; основной механизм замены (во всяком случае, до появления Xen) был основан на сканировании кода операционной системы во время выполнения, поиска некоторых команд и их замене кодом прямого обращения к гипервизору.

В самом деле до появления Xen большинство программ виртуализации стремилось к тому, чтобы виртуальное оборудование с точки зрения программы ничем не отличалось от физического. Таким образом, виртуальные устройства вели себя как физические устройства, эмулировали те же протоколы, а подмена кода гарантировала, что операционная система будет работать без каких-либо изменений. Хотя такой подход гарантирует идеальную совместимость, за него приходится платить дорогую цену в отношении быстродействия. С выходом Xen стало ясно, что отказ от идеальной совместимости позволяет кардинально улучшить быстродействие.

Паравиртуализация

Идея паравиртуализации основана на удалении всех особенностей архитектуры (например, x86), виртуализация которых сопряжена со слишком большими трудностями или затратами, и замене их *паравиртуальными* операциями, взаимодействующими напрямую с уровнем виртуализации. Этот метод был впервые применен в Denali – мониторе виртуальных машин, управляющем специально написанной гостевой операционной системой Ilwaco. Xen делает еще один шаг вперед, обеспечивая возможность запуска паравиртуализованных версий операционных систем массового применения[1].

[1] Конечно, можно сказать, что первой паравиртуализованной операционной системой была VM/370 – операционная система IBM 1960-х годов, предтеча виртуализации. Но поскольку фирма IBM спроектировала специальный набор команд, операционную систему и монитор виртуальных машин, ей не пришлось решать проблемы, возникающие в области современной паравиртуализации.

Паравиртуализация операционной системы требует переписывания всего кода, несовместимого с паравиртуализованной архитектурой. Быстродействие при этом повышается, потому что изменения вносятся разработчиками заранее, а не во время выполнения. Для демонстрации возможностей паравиртуализации группе Xen была необходима операционная система, код которой можно было бы изменять. К счастью, система Linux была доступной, распространялась с открытым кодом и широко применялась на практике. Чтобы система заработала в Xen, в ядре Linux пришлось изменить или добавить всего 2995 строк; это составляет менее 2% кодовой базы Linux для x86. С паравиртуализацией (как и с виртуализацией) все существующие пользовательские приложения продолжают работать без каких-либо изменений, так что общие изменения были не слишком серьезными.

Чтобы добиться паравиртуализации, необходимо написать операционную систему самостоятельно (вариант Denali), изменить существующую операционную систему с открытым кодом (например, Linux или BSD) или убедить разработчиков коммерческой операционной системы, что паравиртуализация их кода – дело стоящее. Исходная исследовательская версия Xen в серии тестов, проведенных с измененной версией Linux, достигла впечатляющего быстродействия, близкого к исходному. Благодаря этому быстродействию, а также тому факту, что проект Xen распространялся с открытым кодом, мы пришли к ситуации, когда разработчики коммерческих операционных систем модифицировали части своего кода, чтобы они более эффективно работали под управлением Xen и других гипервизоров. Еще больший оптимизм внушало нам то, что паравиртуальные операции, разработанные для Xen и других гипервизоров, были стандартизированы в последней версии ядра Linux. Интеграция поддержки Xen (и других гипервизоров) в стандартное ядро только упростила внедрение виртуализации.

Как работает паравиртуализация? Полное описание слишком сложно, чтобы приводить его здесь; в разделе «Библиография» в конце главы представлены статьи, в которых подробно рассматривается этот механизм. А мы ограничимся двумя примерами паравиртуализации: для виртуальной памяти и для виртуальных устройств.

Паравиртуализация операционной системы начинается с того, что операционная система перестает быть самым привилегированным программным кодом, выполняемым на компьютере; эта роль переходит к гипервизору. Большинство процессоров работает как минимум в двух режимах: в режиме *супервизора* и в *пользовательском* режиме. Обычно ядро операционной системы работает в режиме супервизора, но этот режим зарезервирован для Xen, поэтому ядро необходимо изме-

нить для выполнения в пользовательском режиме[1]. Впрочем, даже при работе в пользовательском режиме некоторые операции недопустимы. Это обстоятельство очень важно для защиты процессов друг от друга в обычной операционной системе. Следовательно, ядро должно поручить гипервизору выполнение таких операций от своего лица, используя механизм *гипервызовов* (*hypercalls*). Гипервызов является аналогом системного вызова (от пользовательского процесса к ядру), за исключением того, что он используется для обмена данными между ядром и гипервизором и обычно реализует низкоуровневые операции.

Виртуальная память предотвращает вмешательство процессов в данные или код других процессов. Каждому процессу выделяется виртуальное *адресное пространство*, и процесс может обращаться только к своей выделенной памяти. Ядро отвечает за создание виртуального адресного пространства; для этого оно ведет страничные таблицы, отображающие виртуальные адреса на физические адреса, идентифицирующие фактическое местонахождение данных в физической памяти. При выполнении на виртуальной машине ядро не может бесконтрольно распоряжаться этими таблицами, так как оно может выполнить отображение в память, принадлежащую другой виртуальной машине. Следовательно, Xen должен проверять все обновления страничных таблиц, а ядро должно информировать гипервизор о своих намерениях изменить любую страничную таблицу. Если гипервизору придется участвовать в каждом обновлении страничной таблицы (например, при запуске нового процесса и первом построении его страничных таблиц), такая схема будет крайне неэффективной. Однако, как выясняется, такие случаи относительно редки, и Xen может компенсировать затраты на обращение к гипервизору посредством пакетного объединения запросов или «отсоединения» страничной таблицы на время обновления.

Взгляните на рис. 7.2. Каждая виртуальная машина имеет свою долю общей физической памяти[2]. Тем не менее, эта доля может не быть не-

---

[1] Архитектура x86 имеет четыре уровня привилегий, или *кольца*: уровень 0 – самый привилегированный, уровень 3 обладает наименьшими привилегиями. В 32-разрядной версии Xen работает в кольце 0, паравиртуализированные ядра – в кольце 1, а пользовательские приложения, как обычно – в кольце 3. С другой стороны, в 64-разрядной версии паравиртуализованные ядра работают в кольце 3 из-за различий в аппаратной поддержке сегментации памяти.

[2] Учтите, что Xen не поддерживает необеспеченное выделение памяти (overcommitting), так что выгрузка виртуальных машин не используется. Тем не менее, объем занимаемой памяти виртуальной машины может изменяться при помощи процесса, называемого динамическим перераспределением (ballooning).

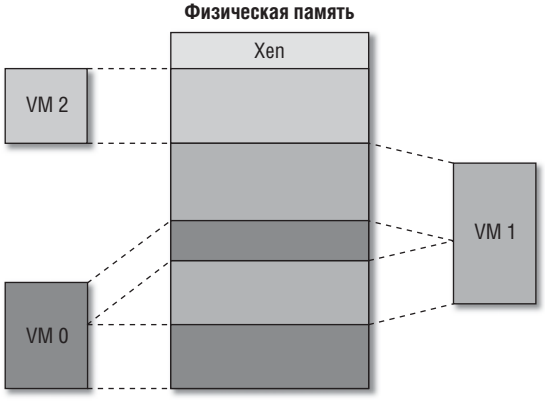


Рис. 7.2. *Структура памяти виртуальной машины*

прерывной, и в большинстве случаев не начинается с физического адреса 0. Следовательно, ядро каждой виртуальной машины должно иметь дело с двумя типами адресов: *физическими* и *псевдофизическими*. Физический адрес соответствует фактическому местонахождению данных в микросхемах памяти, а псевдофизические адреса создают для виртуальной машины иллюзию непрерывного адресного пространства, начинающегося с нуля. Псевдофизические адреса могут быть полезны для некоторых алгоритмов и подсистем, которые зависят от этого условия и без этого нуждались бы в паравиртуализации.

Чтобы виртуальная машина приносила реальную пользу, она должна взаимодействовать с компьютером. Как минимум виртуальной машине необходимы диск (который правильнее называть *блочным устройством*) и сетевая карта[1]. Поскольку большинство операционных систем поддерживает как минимум одно блочное устройство и сетевую карту, возникает искушение эмулировать эти устройства на уровне гипервизора, чтобы иметь возможность использования исходных драйверов. Но программной реализации придется изрядно помучиться, чтобы эмулировать быстродействие реальных устройств, а в эмулируемых моделях устройств придется применить деформации (например, реализацию аппаратных протоколов), излишние и неэффективные для предоставления функций устройства на программном уровне.

[1] Возможно, вы подумали, что для взаимодействия с пользователем также понадобится мышь, клавиатура и видеокарта, но все это может предоставляться клиентом удаленного рабочего стола (таким как VNC). Тем не менее в последних версиях Xen появилась поддержка этих виртуальных устройств.

Как еще это можно сделать?

Стандартный способ виртуализации виртуальной памяти (когда вы не можете изменить операционную систему) основан на использовании *теневых страничных таблиц*. В этом случае гость имеет дело с псевдофизическими адресами (т. е. непрерывными и начинающимися с 0) вместо физических адресов.

Гость поддерживает свои собственные страничные таблицы для этого адресного пространства. Однако эти таблицы не могут использоваться оборудованием, потому что они не соответствуют реальным физическим адресам. По этой причине гипервизор отслеживает обновления этих гостевых страничных таблиц и использует их для конструирования теневой страничной таблицы, преобразующей виртуальные адреса в реальные физические адреса.

Разумеется, этот механизм сопряжен с некоторыми дополнительными затратами, но это необходимо в тех случаях, когда вы не можете изменить операционную систему. Xen использует разновидность этого метода для аппаратных виртуализованных гостей (см. далее в этой главе).

Xen не ограничивается необходимостью поддержки немодифицированных операционных систем, и это позволяет ввести в модель виртуальные блочные и сетевые драйверы. Обе категории драйверов работают сходным образом: они состоят из внешнего драйвера гостевой виртуальной машины и внутреннего драйвера программного обеспечения виртуализации. Два устройства взаимодействуют через циклический буфер – высокопроизводительный механизм передачи больших объемов данных между виртуальными машинами. Это приводит к созданию гибкой иерархической архитектуры (рис. 7.3): внешний драйвер реализует интерфейс сетевого или блочного устройства операционной системы, чтобы виртуальное устройство рассматривалось операционной системой как обычное физическое устройство, а внутренний драйвер соединяет виртуальное устройство с оборудованием. Виртуальное блочное устройство может быть связано с файлом, содержащим образ диска или реального дискового раздела; виртуальное сетевое устройство может быть присоединено к программному сетевому мосту, который в свою очередь присоединяется к реальной сетевой карте. Абстракция циклического буфера гарантирует полное отсутствие привязок между внешним и внутренним драйверами. Один внутренний драйвер может поддерживать внешние драйверы для Linux, BSD или Windows, а один внешний драйвер может использоваться с разными внутренни-

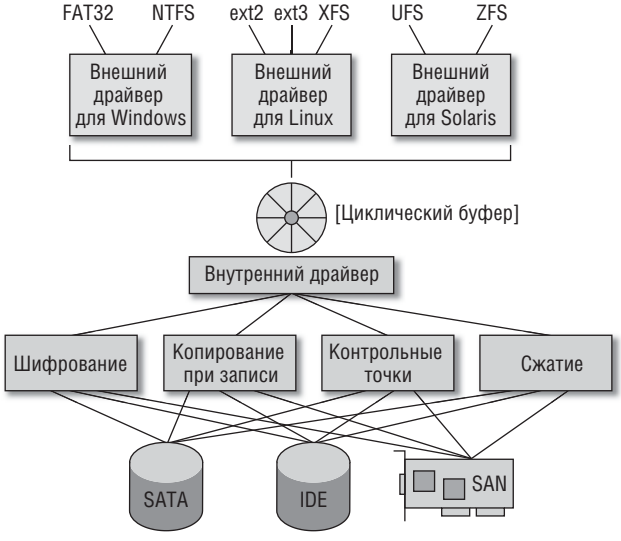


Рис. 7.3. *Архитектура расщепленных блочных устройств*

ми драйверами, что позволяет добавлять такие функции, как копирование при записи, шифрование и сжатие, прозрачно для гостя. По аналогии с протоколом IP, модель разделения устройств Xen может работать на разнообразном оборудовании и поддерживает разнообразных клиентов более высокого уровня, как показано на рис. 7.3.

Паравиртуализация отнюдь не ограничивается этими примерами. Например, само понятие времени изменяется, если операционная система может быть выгружена из процессора, поэтому в Xen вводится концепция виртуального времени, которая обеспечивает предполагаемое поведение операционных систем. Существует много других виртуальных устройств, и в паравиртуализации памяти используется множество иных приемов оптимизации для достижения приемлемого быстродействия. За дополнительной информацией обращайтесь к разделу «Библиография» в конце главы.

Изменяющаяся конфигурация Xen

В традиционном представлении системы на базе Xen обычно изображаются несколькими виртуальными машинами (называемыми в Xen *доменами*), расположенными над гипервизором, который в свою очередь располагается непосредственно над оборудованием (рис. 7.4). При загрузке гипервизор запускает специальный домен, называемый *нуле-*

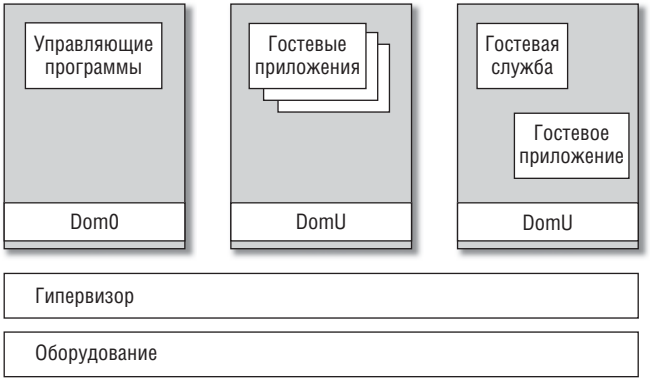


Рис. 7.4. *Системная архитектура Xen*

вым доменом. Нулевой домен обладает специальными привилегиями, которые позволяют ему управлять остальными компонентами системы; его можно сравнить с процессом суперпользователя или администратора в традиционной операционной системе. На рис. 7.4 изображена типичная система на базе Xen с нулевым доменом и несколькими *гостевыми доменами* (в жаргоне Xen используется термин *DomU*), расположенными над гипервизором Xen.

Проектировщики архитектуры Xen стремились по возможности отделить политику от механизма ее реализации. Гипервизор проектировался как тонкая прослойка для управления низкоуровневым оборудованием, выполнявшая функции контрольного монитора, планировщика и системы мультиплексирования доступа к аппаратным устройствам. Но так как гипервизор работает на максимальном уровне привилегий (а ошибка на этом уровне может нарушить работу всей системы), высокоуровневое управление было делегировано в нулевой домен.

Например, при создании новой виртуальной машины основная часть работы выполняется в нулевом домене. С точки зрения гипервизора происходит выделение ресурсов для нового домена, включая часть физической памяти, блок этой памяти распределяется для загрузки операционной системы, после чего домен активизируется. Нулевой домен следит за контролем допуска, настройкой виртуальных устройств и построением образа памяти для нового домена. Такое разделение обязанностей было особенно полезно в процессе разработки, так как управляющие программы гораздо проще отлаживать в нулевом домене, чем в гипервизоре. Более того, оно позволяет включить поддержку разных операционных систем в нулевом домене вместо гипервизора, где любое дополнительное усложнение обычно нежелательно.

Виртуализация типа 2

Xen является примером нативной виртуализации (также называемой виртуализацией типа 1). Альтернативное решение основано на выполнении гипервизора поверх базовой операционной системы (тип 2). В этом случае каждая виртуальная машина фактически становится процессом в базовой операционной системе. Базовая операционная система отвечает за функции управления, выполняемые нулевым доменом в Xen. Гипервизор и управляющие программы напоминают обычные приложения, расположенные над базовой операционной системой (рис. 7.5).

Гипервизоры типа 2 обычно используются в версиях других продуктов виртуализации для «рабочих станций», таких как VMWare Workstation, Parallels Workstation и Microsoft Virtual PC. Основное преимущество данного подхода заключается в том, что установка гипервизора типа 2 сводится к простой установке нового приложения, тогда как установка естественного гипервизора, такого как Xen, больше напоминает установку новой операционной системы. Таким образом, виртуализация типа 2 лучше подходит для неопытных пользователей.

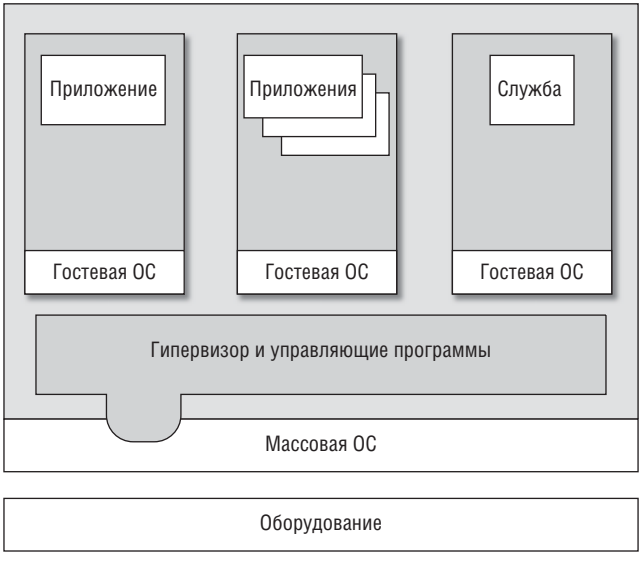


***Рис. 7.5.** Архитектура системы виртуализации типа 2*

С другой стороны, преимущество нативного гипервизора заключается в том, что он может обеспечить более высокое быстродействие, так как нативный гипервизор образует более тонкую программную прослойку, нежели базовая операционная система с гипервизором. Виртуальные машины находятся в полной власти базовой операционной системы, что может привести к ухудшению быстродействия, если параллельно с гипервизором в системе выполняются другие приложения. Напротив, поскольку нулевой домен планируется как обычная виртуальная машина, работающие там приложения не влияют на быстродействие других виртуальных машин.

Виртуальные машины типа 2 обычно используются для виртуализации рабочих столов; например, они позволяют пользователю с Mac OS X запустить Linux в окне на рабочем столе. Такая возможность может пригодиться для запуска приложений, недоступных для основной операционной системы, а в интерактивных приложениях падение быстродействия не столь заметно. Нативная виртуализация лучше подходит для серверной среды, в которой факторы непосредственного быстродействия и предсказуемости чрезвычайно важны.

Как было замечено ранее, проекту Xen помогло наличие операционной системы с открытым кодом, которая стала испытательной моделью для паравиртуализации. Второе преимущество использования Linux заключалось в поддержке широкого диапазона устройств. Xen может поддерживать практически любое устройство, для которого существует драйвер Linux, так как система использует код драйвера Linux. В Xen всегда использовались драйверы Linux для поддержки максимально широкого диапазона оборудования, но между версиями 1.0 и 2.0 природа их использования значительно изменилась.

В Xen 1.0 все виртуальные машины (включая нулевой домен) работали с оборудованием через виртуальные устройства, как описано в предыдущем разделе. Гипервизор отвечал за мультиплексирование доступа к физическому оборудованию и поэтому содержал портированные версии драйверов устройств Linux и внутренние виртуальные драйверы. Хотя такая архитектура упрощала виртуальные машины, она усложняла работу гипервизора и возлагала бремя поддержки новых драйверов на группу разработки Xen.

На рис. 7.6 представлены изменения архитектуры устройств между Xen 1.0 и 2.0. В версии 1.0 виртуальные внутренние драйверы были

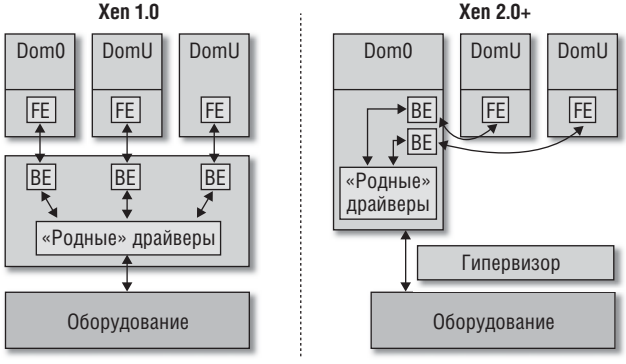


Рис. 7.6. *Изменения в архитектуре устройств между Xen 1.0 и 2.0*

реализованы в гипервизоре: все домены, включая нулевой, обращались к оборудованию через эти устройства. В версии 2.0 гипервизор был облегчен, а нулевой домен получил доступ ко всему оборудованию через «родные» драйверы. Соответственно внутренние драйверы переместились в нулевой домен.

В ходе разработки Xen 2.0 архитектура устройств была полностью переработана: «родные» драйверы устройств и виртуальные внутренние драйверы были выведены из гипервизора в нулевой домен[1]. Теперь для передачи данных между внешними и внутренними драйверами используются *каналы устройств*, которые обеспечивают эффективный и защищенный обмен данными между доменами. Благодаря каналам устройств виртуальные устройства Xen по быстродействию почти не уступают физическим устройствам. Их быстродействие базируется на двух принципах: передаче без копирования и асинхронном оповещении.

Взгляните на рис. 7.7. Эта диаграмма показывает, как используется разделенное устройство. Гость представляет внешнему драйверу страницу памяти – либо с данными для записи, либо для хранения читаемых данных (1). Внешний драйвер помещает в следующий доступный слот общего циклического буфера запрос, содержащий ссылку на предоставленную страницу (2), и приказывает гипервизору оповестить драйверный домен о наличии ожидающего запроса (3). Внутренний драйвер активизируется и отображает предоставленную страницу в свое адресное пространство (4), чтобы оборудование могло взаимодейство-

[1] Более того, новая архитектура позволяет любой авторизованной виртуальной машине работать с оборудованием, то есть выполнять функции *драйверного домена*.

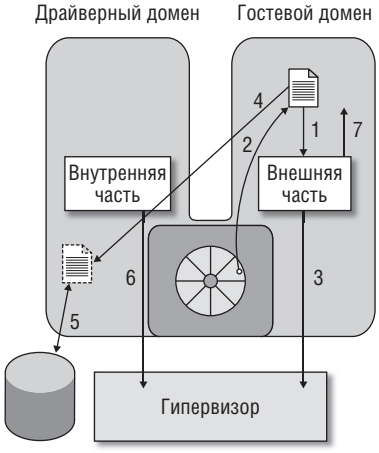


Рис. 7.7. *Строение разделенного устройства*

вать с ним по каналу DMA (5). Наконец, внутренний драйвер оповещает внешний о завершении запроса (6), а внешний драйвер оповещает гостевое приложение (7).

Копирование данных с участием процессора является затратной операцией. По этой причине были разработаны такие технологии, как *DMA* (Direct Memory Access), предназначенные для перемещения данных в памяти без участия процессора. Но когда данные приходится перемещать между адресными пространствами, Xen приходится принимать особые меры для предотвращения копирования. Xen поддерживает механизм общего доступа к памяти, называемый *таблицами доступа* (*grant tables*); каждая виртуальная машина ведет таблицу, определяющую, какие из ее страниц могут быть доступны для других виртуальных машин. Для обращения другой виртуальной машине передается индекс элемента этой таблицы, называемый *ссылкой доступа*. Гипервизор следит за тем, чтобы обращение по ссылке доступа было возможно только для законного получателя, что способствует изоляции памяти. Ссылки доступа передаются по каналу устройства и используются для отображения буферов отправки или передачи данных в памяти.

При передаче нового запроса или ответа отправитель должен оповестить получателя. В традиционных схемах использовались синхронные оповещения (аналоги вызовов функций), то есть работа отправителя блокируется до того момента, когда он будет знать, что оповещение было получено. Как показано на рис. 7.8, такой режим работы ведет к низкой производительности, особенно если доступен всего один

процессор. Вместо этого Xen использует *каналы событий* для передачи асинхронных оповещений. Каналы событий реализуют виртуальные прерывания, но виртуальное прерывание обрабатывается только при следующем выделении кванта времени целевому домену. Таким образом, запрашивающая сторона может сгенерировать несколько запросов, каждый раз активизируя канал событий, прежде чем целевой домен получит возможность на них отреагировать. Затем при следующем выделении процессора целевой домен может обработать сразу несколько запросов и отправить ответы (снова асинхронно).

Взгляните на рис. 7.8. С синхронными оповещениями внешнему драйверу приходится ожидать, пока внутренний драйвер завершит свою работу, прежде чем выдавать следующий запрос. А это означает, что он должен сначала дождаться выделения процессора внутреннему домену, а затем – повторного выделения процессора внешнему домену. С другой стороны, при асинхронных оповещениях внешний драйвер может отправить столько запросов, сколько успеет за выделенное ему время, после чего внутренний драйвер отправляет максимально возможное количество ответов. Такая схема существенно повышает производительность обработки запросов.

Конечно, если сосредоточить слишком много функциональности в нулевом домене, он превратится в единую точку отказа (т. е. «уязвимое

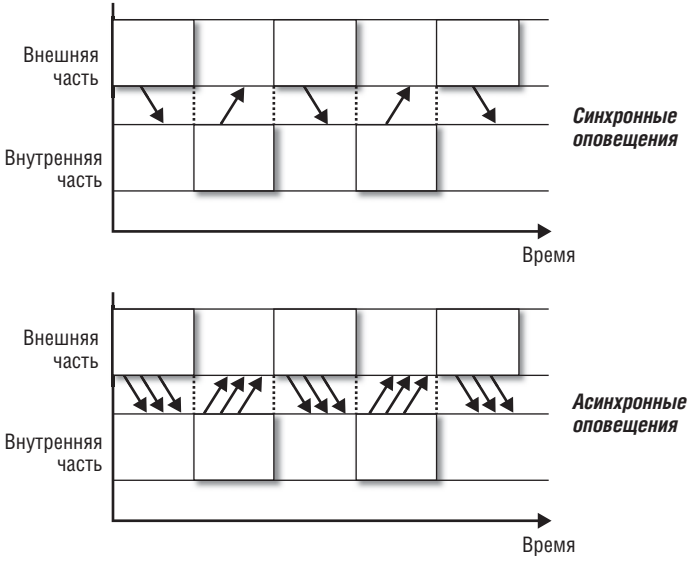


Рис. 7.8. *Преимущества асинхронных оповещений*

звено»). Это особенно справедливо по отношению к сбоям устройств, способным вывести из строя всю операционную систему (а вместе с ней и всю виртуализованную систему). По этой причине в Xen появилась поддержка драйверных доменов, которым нулевой домен может делегировать контроль за одним или несколькими устройствами. Задача решается простым размещением внутреннего драйвера в драйверном домене, которому предоставляются некоторые привилегии ввода/вывода. Если в драйвере произойдет сбой, то он будет изолирован в драйверном домене, который может быть перезапущен без вреда для системы или клиентского домена.

Эта модель была применена и к другим компонентам нулевого домена. В новейших версиях Xen появились *домены-заглушки* (*stub domain*), обеспечивающие поддержку устройств для «аппаратно-виртуализованных» доменов (см. следующий раздел). Перемещение кода в изолированные домены улучшает изоляцию и повышает надежность системы, а иногда, как ни удивительно, улучшает непосредственную производительность. Возможно, по мере развития из нулевого домена будут выведены и другие функции, особенно если это приведет к улучшению безопасности.

Изменения в оборудовании – изменения в Xen

До настоящего момента наше обсуждение было сосредоточено на теме паравиртуализации. Однако между выходом Xen версий 2.0 и 3.0 фирмы Intel и AMD реализовали в своих процессорах разную, но концептуально сходную поддержку *аппаратных виртуальных машин*. Появилась возможность выполнения немодифицированных операционных систем, включая Microsoft Windows и Linux, в виртуальных машинах. Означает ли это конец паравиртуализации?

Прежде всего, давайте разберемся, как реализованы виртуальные машины. Как Intel, так и AMD вводят новый режим (*привилегированный режим* у Intel, *гостевой режим* у AMD), в котором при попытке выполнения привилегированных операций, даже на самом высоком (виртуальном) уровне привилегий, происходит исключение, оповещающее гипервизор. Таким образом, отпала необходимость в сканировании кода и замене опасных команд (во время выполнения или заранее, посредством паравиртуализации). Гипервизор может использовать теневые страничные таблицы, чтобы создать у виртуальной машины иллюзию непрерывной памяти, а затем перехватывать операции ввода/вывода для эмуляции физических устройств.

В Xen поддержка аппаратных виртуальных машин появилась в версии 3.0. Разработка на условиях открытого кода значительно упрости-

ла переход. Так как Xen является проектом с открытым кодом, разработчики из Intel и AMD могли вносить низкоуровневый код для поддержки новых процессоров. Более того, благодаря лицензии GPL проект Xen мог включать код из других проектов с открытым кодом. Например, поддержка новых аппаратных виртуальных машин требовала эмуляции BIOS и аппаратных устройств; самостоятельная реализация потребовала бы огромных усилий со стороны разработчиков. К счастью, разработчики Xen смогли воспользоваться BIOS с открытым кодом из проекта Bochs и эмуляцией устройств из QEMU.

Резонно спросить, что же произойдет с хваленными преимуществами паравиртуализации? Действительно ли все эти эмулируемые устройства, теневые страничные таблицы, дополнительные исключения приведут к падению быстродействия? Конечно, операционные системы с наивной аппаратной виртуализацией работают заметно хуже паравиртуализованных операционных систем, но здесь необходимо учитывать два фактора.

Во-первых, фирмы-разработчики процессоров постоянно создают новые функции, оптимизирующие виртуализацию. Если блок управления памятью *MMU* (Memory Management Unit) позволяет программистам иметь дело с виртуальными адресами вместо физических, блок IOMMU выполняет ту же функцию для устройств ввода/вывода. IOMMU предоставляет виртуальной машине (аппаратно-виртуализованной или паравиртуализованной) прямой безопасный доступ к устройству (рис. 7.9). Обычная проблема при прямом доступе к оборудованию со стороны виртуальных машин заключается в том, что передача DMA может выполняться многими устройствами, и без IOMMU появляется возможность чтения или перезаписи памяти других виртуальных машин. IOMMU гарантирует, что во время активности конкретной виртуальной машины для DMA будет доступна только память, принадлежащая этой машине.

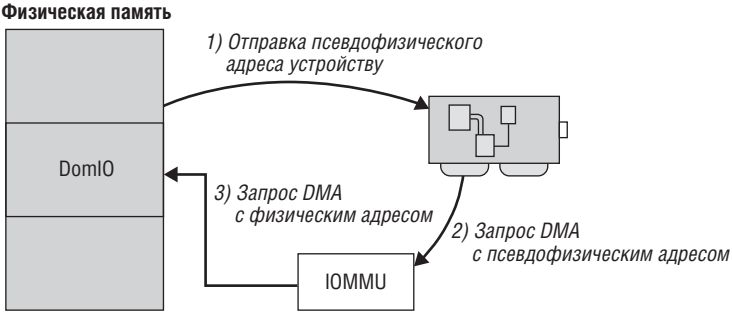


Рис. 7.9. *Прямой доступ к устройству с использованием IOMMU*

На рис. 7.9 показан упрощенный запрос DMA от виртуальной машины (DomIO) с использованием IOMMU. При взаимодействии с устройством (1) аппаратный драйвер использует псевдофизические (привязанные к конкретной виртуальной машине) адреса. Устройство выдает запросы DMA, используя эти адреса (2), а IOMMU (по страничным таблицам ввода/вывода, настроенным гипервизором) преобразует их в физические адреса (3). IOMMU также пресекает любые попытки виртуальных машин обратиться к памяти, им не принадлежащей.

Совершенствование аппаратной поддержки управления памятью также может снять необходимость в теневых страничных таблицах[1]. Как у AMD, так и у Intel появились технологии (Rapid Virtualization Indexing и Enhanced Page Tables соответственно), выполняющие преобразования между псевдофизическими и физическими адресами. Таким образом, гипервизору не обязательно создавать теневые страничные таблицы, поскольку все преобразование выполняется на аппаратном уровне.

Конечно, гораздо менее затратное решение заключается в том, чтобы применить уроки паравиртуализации к немодифицированным гостевым операционным системам. Изменить основные части операционной системы невозможно, но мы можем добавить драйверы устройств; кроме того, Xen может изменить виртуальное оборудование, на котором работает операционная система. В этом отношении эмулируемое оборудование предоставляет устройство платформы Xen, которое воспринимается немодифицированными гостевыми операционными системами как устройство PCI и предоставляет доступ к виртуальной платформе. Затем можно написать внешние драйверы устройств для немодифицированных операционных систем, которые работают так же, как внешние драйверы в паравиртуализованных операционных системах. Тем самым в аппаратных виртуальных машинах будет достигнуто быстродействие ввода/вывода, сравнимое с быстродействием паравиртуализации.

Во вводном описании паравиртуализации ранее в этой главе мы упоминали о том, что заставить массовую операционную систему работать в качестве паравиртуализованного гостя можно только двумя способами: либо модифицировать ее самостоятельно, либо убедить разработчиков коммерческой операционной системы в том, что ее стоит переделать. Как свидетельство успеха паравиртуализации, компания Microsoft включила в Windows Server 2008 так называемые *усовершенствования* (*enlightenments*), которые повышают быстродействие управления

[1] Следует заметить, что реализация теневых страничных таблиц в Xen хорошо оптимизирована и обеспечивает приемлемое быстродействие, и все же слегка уступает по быстродействию паравиртуализованным страничным таблицам.

памятью при запуске на виртуальной машине. Эти усовершенствования являются аналогами паравиртуализованных операций, так как они используют гипервызовы для оповещения гипервизора о текущей операции.

Эмуляция против виртуализации

Последняя версия Xen включает код из проектов Bochs и QEMU; оба проекта относятся к *эмуляторам*. Чем эмуляция отличается от виртуализации и как они сочетаются друг с другом?

Bochs обеспечивает программную реализацию с открытым кодом семейства процессоров x86, а также поддерживающего оборудования. QEMU эмулирует несколько архитектур, включая x86. Оба эмулятора могут использоваться для выполнения немодифицированных операционных систем и приложений x86. Более того, поскольку они включают полную реализацию оборудования (в том числе и процессора), появляется возможность их выполнения на оборудовании с несовместимым набором команд.

Виртуализированные и эмулируемые системы различаются по способу выполнения команд. В виртуализированной системе приложения и большая часть операционной системы выполняются непосредственно процессором, тогда как в эмулируемой системе эмулятор должен имитировать или транслировать каждую команду для ее выполнения. Следовательно, работа эмулятора сопряжена с бóльшими затратами, чем работа монитора виртуальных машин для той же платформы[1].

Тем не менее, несмотря на использование фрагментов Bochs и QEMU, в аппаратных виртуальных машинах Xen используется виртуализация, а не эмуляция. Код Bochs предоставляет систему BIOS, поддерживающую процесс загрузки, а QEMU – эмулируемые драйверы для широкого спектра стандартных устройств. Однако эти блоки кода активизируются только при запуске системы и при попытке выполнения операции ввода/вывода. Большая часть других команд выполняется непосредственно на процессоре.

[1] Модуль ядра Linux KQEMU позволяет выполнять код пользовательского режима (и часть кода режима ядра) непосредственно на процессоре. При совпадении базовой и целевой платформ достигается существенное ускорение эмуляции. Результат представляет собой гибрид эмуляции и виртуализации.

Уроки Xen

Оглядываясь назад, мы можем извлечь из Xen два основных урока: важность паравиртуализации и преимущества разработки с открытым кодом.

Паравиртуализация

Первое и самое главное – это успех паравиртуализации. Знаменитая цитата напоминает нам:

> *Любая проблема в области компьютерных технологий может быть решена введением дополнительного уровня абстракции. Но при этом обычно возникает новая проблема.*
>
> Дэвид Уилер

Виртуализация – всего лишь разновидность абстракции. Несмотря на то, что современные компьютеры обладают аппаратной поддержкой виртуализации, было бы наивно рассчитывать только на эту поддержку – это приведет к низкому быстродействию. Аналогичные проблемы возникают при любом наивном использовании виртуализации.

Например, виртуальная память использует жесткий диск для создания иллюзии огромного объема свободной памяти. Но если вы напишете программу, которая попытается использовать всю виртуальную память так, как если бы она была реальной, физической памятью, ее быстродействие будет ужасным. В таком случае стоит рассмотреть возможность «паравиртуализации» программы – учесть наличие физических ограничений в программном коде, изменить алгоритмы и структуры данных, чтобы они эффективно работали в сочетании с системой виртуальной памяти.

В контексте операционных систем проект Xen показал, что паравиртуализация – будь то добавление виртуального драйвера, изменение всего кода операционной системы, или добавление усовершенствований для повышения производительности в отдельных областях – является важным фактором повышения быстродействия при выполнении в виртуальной среде.

Разработка с открытым кодом

Возможно, самым смелым решением в ходе разработки Xen был выбор модели распространения с открытым кодом, тогда как другие гипервизоры были доступны только в виде коммерческих программных продуктов.

Это решение определенно пошло на пользу Xen из-за огромного количества программных продуктов, код которых можно было использовать, от ядра Linux и эмулятора QEMU до крошечной программы, выводящей логотип Xen во время загрузки[1]. Без этого кода разработчикам Xen пришлось бы потратить огромные усилия на повторную реализацию. Обновление включаемых проектов приносит пользу Xen, а исправления, созданные разработчиками Xen, приносят пользу другим проектам.

В работе над проектом Xen, начатым по совместительству одним аспирантом Кембриджского университета, сейчас работает свыше 100 участников из многих стран мира. Ценнейший вклад внесли сотрудники Intel и AMD, предоставившие большую часть кода поддержки аппаратных виртуальных машин. В результате Xen стал одним из первых гипервизоров, поддерживающих эти расширения новых процессоров.

Более того, так как Xen распространяется бесплатно, он был включен в ряд других проектов. Многие известные дистрибутивы Linux, такие как Debian, Red Hat, SUSE и Ubuntu, сейчас включают пакеты Xen, а также передают свой код проекту вместе с полезными инструментами для использования Xen. Некоторые участники не пожалели усилий на портирование Xen для других архитектур, а также портирование других операционных систем для прямого выполнения под управлением гипервизора. В частности, код Xen использовался для запуска паравиртуализованных OpenSolaris, FreeBSD и NetBSD. Сейчас Xen работает на архитектуре Itanium, причем идет работа над его портированием на процессор ARM. Последнее особенно интересно, потому что это позволит Xen работать на «нетрадиционных» устройствах, таких как мобильные телефоны.

Заглядывая в будущее, можно предсказать, что некоторые из самых интересных применений Xen следует искать в исследовательском сообществе. Проект Xen был представлен на Симпозиуме по принципам операционных систем (SOSP) в 2003 году, он заложил основу для разнообразных исследований – как в рамках исходной исследовательской группы, так и за ее пределами. Одна из самых ранних статей о Xen была написана в Университете Кларксона, где группа исследователей повторила результаты из данных SOSP. Авторы отметили, что программное обеспечение с открытым кодом способствует развитию теории вычислительных систем, потому что оно делает возможным повторные исследования, которые в свою очередь усиливают любые утверждения относительно производительности или других характеристик. Более

[1] Figlet: *http://www.figlet.org*

современные исследования напрямую привели к появлению новых интересных возможностей в Xen. Конкретным примером служит механизм *оперативной миграции*, позволяющий перемещать виртуальные машины между физическими компьютерами за ничтожно малый период бездействия. Эта возможность была подробно описана в статье в 2005 году, после чего была реализована в Xen версии 2.0.

Библиография

Эта глава дает лишь самое отдаленное представление о проекте Xen. За дополнительной информацией следует обращаться к соответствующим научным статьям.

Первые две статьи подробно описывают архитектуру Xen 1.0 и 2.0 соответственно:

Barham, Paul, et al. «Xen and the art of virtualization», *Proceedings of the 19th ACM Symposium on Operating System Principles*, October, 2003.

Fraser, Keir, et al. «Safe hardware access with the Xen virtual machine monitor», *Proceedings of the 1st OASIS Workshop*, October, 2004.

В следующих статьях подробно описываются некоторые новые технологии чипсетов и процессоров, разработанные для поддержки виртуализации:

Ben-Yehuda, Muli, et al. «Using IOMMUs for virtualization in Linux and Xen», *Proceedings of the 2006 Ottawa Linux Symposium*, July, 2006.

Dong, Yaozu, et al. «Extending Xen with Intel virtualization technology», *Intel® Technology Journal*, August, 2006.

Наконец, проект Xen находится в активной разработке и постоянном развитии. Если вы хотите быть в курсе всех новых событий, загрузите исходный код и включайтесь в список рассылки. И то, и другое можно сделать на сайте *http://www.xen.org/*.

Принципы и свойства	Структуры
✓ Гибкость	✓ Модуль
✓ Концептуальная целостность	✓ Зависимость
✓ Возможность независимого изменения	✓ Обработка
✓ Автоматическое распространение	✓ Доступ к данным
Удобство построения	
✓ Адаптация к росту	
✓ Сопротивление энтропии	

8

Guardian: отказоустойчивая операционная система

Грег Лехи

В архитектуре нет ничего нового. Здания строятся уже тысячи лет, и самым красивым построенным зданиям уже тоже тысячи лет. Конечно, история компьютеров намного короче, но и в ней можно встретить множество примеров красивых архитектур. Как и в области зодчества, стиль не всегда выдерживает проверку временем. В этой главе я опишу одну такую архитектуру и постараюсь объяснить, почему она оказала столь незначительное влияние в своей области.

Guardian – операционная система для серии отказоустойчивых компьютеров «NonStop» фирмы Tandem. Система проектировалась параллельно с оборудованием, обеспечивающим отказоустойчивость с минимальными дополнительными затратами.

В этой главе описывается исходная машина Tandem, которая проектировалась с 1974 до 1976 года и продавалась с 1976 до 1982 года. Изначально она называлась «Tandem/16», но после появления наследника «NonStop II» ее задним числом переименовали в «NonStop I». В Tandem термин «T/16» часто использовался как для обозначения самой системы, так и, позднее, для обозначения архитектуры.

Я работал с оборудованием Tandem на штатной должности с 1977 по 1991 год. Работать с машиной Tandem было занятно и необычно. В этой главе мне хотелось бы оживить некоторые чувства, которые программисты испытывали к этой машине. T/16 была отказоустойчивой машиной, но это была не единственная ее характеристика. В этом обсуждении упоминаются многие аспекты, не связанные напрямую с отказоустойчивостью – более того, пара аспектов противоречит ей! Итак, приготовьтесь к путешествию в прошлое – примерно в 1980 год. Все началось с одного из рекламных лозунгов Tandem.

Tandem/16: когда-нибудь все компьютеры будут такими

Tandem описывает свои машины как отдельные компьютеры с несколькими процессорами, но с точки зрения XXI века они больше напоминают компьютерную сеть, работающую как отдельный компьютер. В частности, каждый процессор работает почти полностью независимо от других, а система может восстановиться после сбоя любого компонента, включая процессоры. Принципиальное отличие от традиционных процессорных сетей заключалось в том, что вся система работает на базе одного образа ядра.

Оборудование

Оборудование Tandem проектировалось так, чтобы в нем не существовало потенциальных «единых точек отказа»; сбой любого отдельного компонента системы, аппаратного или программного, не приводил к сбою всей системы. Кроме того, система проектировалась с учетом *корректного сокращения функциональности*. В большинстве случаев система как единое целое сможет продолжить работу даже при множественных сбоях, хотя это сильно зависит от природы отдельных сбоев.

Первое следствие такого архитектурного подхода заключается в том, что каждый компонент должен присутствовать как минимум в двух экземплярах, на случай сбоя одного экземпляра. В частности, это означает, что система должна быть оснащена как минимум двумя процессорами.

Но как связать эти два процессора? Традиционный способ – и тогда, и сейчас – был основан на взаимодействии через общую память. В Tandem такая схема называлась *сильносвязанными мультипроцессорами*. Но если процессоры используют общую память, то эта память может стать единой точкой отказа.

Дублирование памяти теоретически возможно (и оно было реализовано в одной из более поздних архитектур Tandem), но оно обходится очень дорого и создает серьезные проблемы с синхронизацией. Вместо этого на аппаратном уровне в Tandem было выбрано другое решение: пара скоростных параллельных межпроцессорных шин, или *IPB*, иногда также называемых *Dynabus* и передающих данные между отдельными процессорами. Такая архитектура иногда называется *слабосвязанными мультипроцессорами*.

Конечно, компьютер состоит не только из процессора. В частности, очень важную роль играет система ввода/вывода и организация хранения данных. Основным принципом архитектуры также было дублирование оборудования; вскоре мы вернемся к этой теме.

Примерный вид итоговой архитектуры показан на рис. 8.1 – так называемой *диаграмме Мэки*, от фамилии Дэйва Мэки (Dave Mackie), вице-президента Tandem.

Это могло бы легко привести к удвоению стоимости системы, как в системах «оперативного резерва», где один компонент присутствует

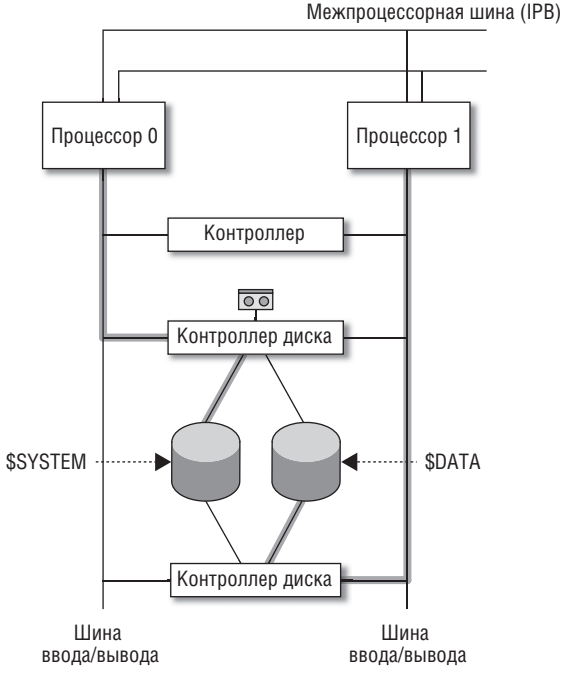


Рис. 8.1. *Диаграмма Мэки*

только в ожидании сбоя своего «напарника». Для более дорогих компонентов, таких как процессоры, фирма Tandem выбрала другой подход. В T/16 каждый процессор активен, а функция оперативного резерва обеспечивается процессами операционной системы.

Диагностика

Когда на каком-то компоненте происходит сбой, операционная система должна узнать об этом. Во многих случаях сомневаться не приходится; при катастрофическом сбое компонент полностью перестает реагировать на запросы. Однако нередко сбойный компонент продолжает работать, хотя выдает неверные результаты.

Решение, выбранное Tandem, не отличается ни элегантностью, ни эффективностью. Программное обеспечение проектируется «параноидально»; едва заподозрив, что где-то возникли проблемы, операционная система останавливает процессор – есть другой, который может взять на себя нагрузку. Если контроллер диска возвращает недопустимый код состояния, он отключается – операция может быть продолжена другим контроллером. Но особенно коварные сбои могут остаться незамеченными, и в отдельных случаях это приводит к повреждению данных.

Конечно, одного лишь сбоя процессора недостаточно для переключения; другие процессоры должны узнать о произошедшем сбое. Проблема решалась по схеме «стража»: каждый процессор через каждые 1,2 секунды рассылает сообщение («я жив») по обеим шинам. Если процессор не получил два последовательных сообщения от другого процессора, он считает, что на том процессоре произошел сбой. Если процессоры совместно используют процессы или ресурсы ввода/вывода, то процессор, обнаруживший сбой, берет их под свой контроль.

Восстановление

Просто отключить сбойный компонент недостаточно; для обеспечения отказоустойчивости и оптимальной производительности его необходимо как можно быстрее вернуть в исходное состояние – и, конечно, без отключения других компонентов.

Процедура восстановления зависит от компонента и природы сбоя. Если на одном процессоре произошел сбой операционной системы (возможно, намеренный), то ее можно перезагрузить с возвращением в работоспособное состояние. В стандартном варианте загрузки системы первый процессор загружается с диска, после чего все остальные процессоры загружаются через IPB. Сбойные процессоры также перезагружаются через IPB.

С другой стороны, если оборудование вышло из строя, его необходимо заменить. Все компоненты системы поддерживают *оперативную замену*: их можно отключать и заменять в работающей системе с включенным питанием. Если на процессоре произойдет сбой из-за аппаратной проблемы, то на компьютере заменяется соответствующая плата, а процессор перезагружается по шине, как обычно.

Механическое строение

Система спроектирована с расчетом на минимальное количество плат, поэтому все платы очень велики – около 50×50 см. Все платы используют маломощные ТТЛШ-схемы.

Центральный процессор состоит из двух компонентов: вычислительного блока и MEMPPU. Блок MEMPPU содержит интерфейс для работы с памятью, включая логику виртуальной памяти и интерфейс для работы с шиной ввода/вывода. T/16 может оснащаться до 512 килослов (1 Мбайт) полупроводниковой памяти или 256 килослов памяти на магнитных сердечниках. Платы памяти существуют в трех вариантах: 32 килослов на магнитных сердечниках, 96 и 192 килослов полупроводниковой памяти. Это означает, что при заполнении всех плат невозможно получить ровно 1 Мбайт полупроводниковой памяти. Память на магнитных сердечниках защищена проверкой четности слов, а полупроводниковая память использует защиту ECC, которая способна исправлять ошибки в одном бите и выявлять ошибки в двух битах.

Процессорные стенды имеют высоту около 6 футов; на них устанавливаются четыре процессора с полупроводниковой памятью или четыре процессора с памятью на магнитных сердечниках. Процессоры располагаются в верхней части стенда, а контроллеры ввода/вывода занимают вторую полку непосредственно под ними. Ниже расположены охлаждающие вентиляторы, а в самом низу стойки размещаются аккумуляторы для сохранения содержимого памяти во время сбоев питания.

В большинстве конфигураций присутствует второй стенд с ленточным накопителем. Дисководы представляют собой автономные 14-дюймовые устройства. Также имеется системная консоль – печатный терминал DEC LA-36.

Архитектура процессора

Процессор – разработанная по специальному заказу ТТЛ-схема, обладающая значительным сходством с Hewlett-Packard 3000. Он оснащен виртуальной памятью с размером страницы 2 Кбайт, стековым набором команд и 16-разрядными командами фиксированного размера.

Физическая производительность каждого процессора составляет около 0,8 MIPS, что составляет около 13 MIPS в полностью оснащенной 16-процессорной системе.

Адресация памяти

T/16 – 16-разрядная машина, а адресное пространство ограничено 16-разрядными адресами. Даже в конце 1970-х годов это создавало проблемы, для решения которых фирма Tandem предоставила сразу четыре адресных пространства:

Пользовательский код

Адресное пространство для хранения исполняемого кода доступно только для чтения и используется совместно всеми процессами. Вследствие особенностей архитектуры (отдельная память для каждого процессора) совместное использование кода возможно только на одном процессоре.

Пользовательские данные

Пространство данных пользовательских процессов.

Системный код

Код ядра.

Системные данные

Пространство данных ядра.

В любой момент времени доступно только одно пространство данных и одно пространство кода (с одним исключением). Они задаются в *регистре окружения* E (Environment), содержащем набор флагов с описанием текущего состояния процессора (рис. 8.2).

Рис. 8.2. *Регистр E*

Бит SD определяет пространство данных, а бит SC определяет пространство кода. Исключением из этого правила является SG-относительный режим адресации: в нем всегда адресуются системные данные.

В соответствии с поставленной целью обеспечения надежности и целостности данных бит trap в регистре E разрешает, среди прочего, системные прерывания при математическом переполнении. Также существуют «логические» эквиваленты математических команд, которые не устанавливают коды состояния.

Процессор содержит аппаратный стек, адресуемый двумя регистрами: регистр S (указатель стека) и регистр L (указатель на текущий кадр стека). Регистр L, ссылающийся на базу текущего кадра, является относительно новой идеей. В отличие от регистра S, он не изменяется во время выполнения процедуры[1]. Вследствие выбранной схемы адресации стек ограничивается первыми 32 Кбайт текущего пространства данных и, в отличие от некоторых других машин, он растет в направлении увеличения адресов[2].

Кроме аппаратного стека имеется также стек регистров из восьми 16-разрядных слов. Регистры пронумерованы от R0 до R7, но в наборе команд они используются как циклический стек, вершина которого определяется битами RP регистра E. В следующем примере биты RP содержат значение 3; таким образом, вершиной стека является регистр R3 (он называется регистром A) (рис.8.3).

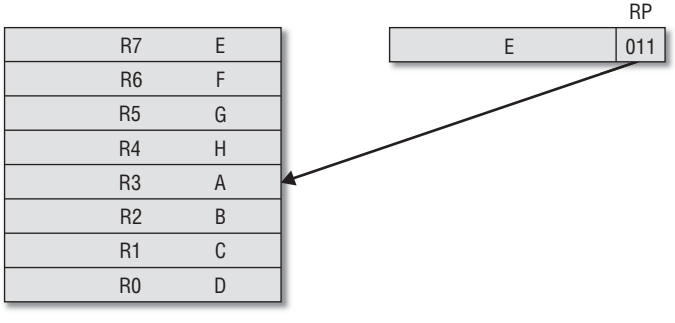


Рис. 8.3. *Стек регистров*

[1] Этот регистр является аналогом регистра BP (Base Pointer), используемого в большинстве процессоров XXI века.

[2] В 1970-х годах стеки были относительно новой идеей. Поддержка стеков у машины Tandem, как и у ее предшественника HP 3000, значительно превосходила поддержку стеков у DEC PDP-11, наиболее заметной стековой машины того времени.

Если предположить, что стек регистров «пуст» в начале выполнения, типичная последовательность команд будет выглядеть так:

```
LOAD    var^a          -- занести var^a в стек (R0)
LOAD    var^b          -- занести var^b в стек (R1)
ADD                    -- сложить A и B (R1 и R0),
                       -- с сохранением результата в R0 (A)
STOR    var^c          -- сохранить A в var^c
```

Все команды занимают 16 бит, поэтому для поля адреса остается не так уж много места: всего 9 бит. Для решения этой проблемы Tandem использует адресацию со смещением от группы регистров (рис. 8.4).

Адресоваться напрямую (то есть без применения косвенной адресации) могут только следующие области памяти:

- Первые 256 слов текущего пространства данных – режим `G` (Global). Область часто используется для косвенных указателей.
- Первые 128 слов положительного смещения от регистра `L` (режим `L+`). Это локальные переменные текущего вызова процедуры (в языке C они называются *автоматическими* переменными).

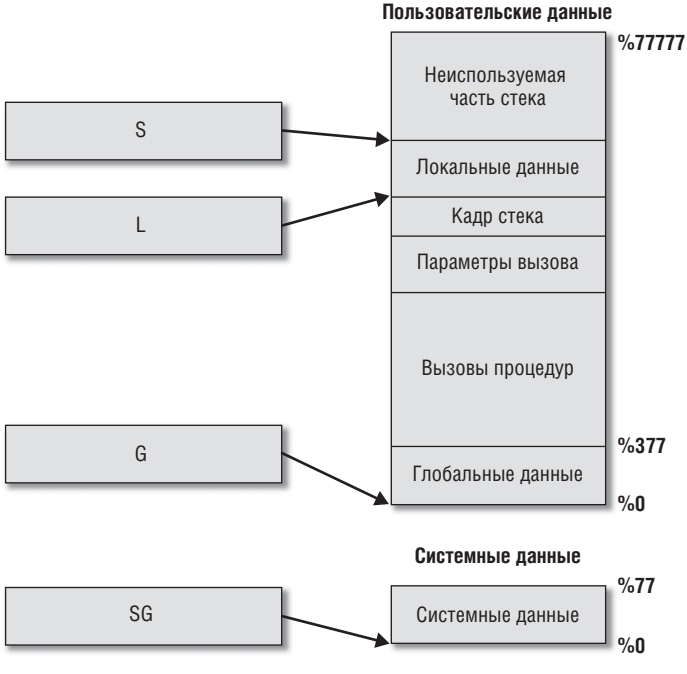


***Рис. 8.4.** Адресация памяти*

- Первые 64 слова системных данных (режим SG+). Системные вызовы выполняются в пространстве пользовательских данных, поэтому процессору необходим механизм обращения к системным данным со стороны привилегированных процедур. Для непривилегированных процедур эти данные недоступны даже для чтения.
- Первые 32 слова под текущим значением регистра L. В эту область входит кадр стека вызова (3 слова) и до 29 слов параметров, передаваемых процедуре.
- Первые 32 слова под вершиной стека (адресация S-). Используются для субпроцедур (процедур, определяемых в других процедурах, которые вызываются без выхода из кадра стека). Этот режим адресации используется для работы как с локальными переменными, так и с параметрами субпроцедур.

Эти режимы адресации кодируются в нескольких начальных битах поля адреса команды (рис. 8.5).

Префикс % обозначает восьмеричные числа – например, %377 (255 в десятичной, или 7F в шестнадцатеричной системе). Tandem не использует шестнадцатеричную запись.

Формат команды также поддерживает одноуровневую косвенную адресацию: если в команде установлен бит I, то прочитанное слово данных интерпретируется как адрес операнда, который должен находиться в том же адресном пространстве. Разрядность адресного пространства и слова данных составляет 16 бит, поэтому многоуровневая косвенная адресация невозможна.

Недостаток этой реализации заключается в том, что единицей данных в ней является 16-разрядное слово, а не байт. Набор команд также содержит «байтовые команды» с другим методом адресации: младший

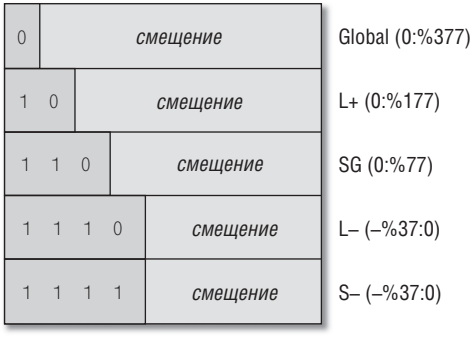


***Рис. 8.5.** Формат адреса*

бит адреса задает байт слова, а оставшаяся часть адреса – младшие 15 бит адреса слова. Для обращения к данным такая схема ограничивает адресацию первыми 32 Кбайт пространства данных; для обращения к коду доступ ограничивается половиной адресного пространства, к которой принадлежит текущая команда. В результате появилось ограничение, согласно которому процедура не может выходить за границу 32 Кбайт в пространстве кода.

Также существуют две команды, `LWP` (Load Word from Program) и `LBP` (Load Byte from Program), которые позволяют обращаться к данным в текущем пространстве кода.

Вызовы процедур

Модель программирования Tandem многое позаимствовала из мира Algol и Pascal, поэтому термином «функция» в ней обозначаются функции, возвращающие значение, а термином «процедура» – функции без возвращаемого значения. Для вызова процедур используются две команды: `PCAL` для процедур в текущем пространстве кода и `SCAL` для процедур в системном пространстве кода. Команда `SCAL` является аналогом *системных функций* в других архитектурах.

Все вызовы осуществляются посредством косвенной адресации по таблице точек входа в процедуры *PEP* (Procedure Entry Point), занимающей первые 512 слов каждого пространства кода. Последние 9 разрядов команды PCAL или SCAL содержат индекс в этой таблице.

У такого механизма есть как преимущества, так и недостатки: ядро использует такой же метод вызова функций, как и пользовательский код, что упрощает программирование и позволяет перемещать код между ядром и пользовательским пространством. С другой стороны, команда `SCAL` хотя бы теоретически позволяет любой пользовательской программе вызвать любую функцию ядра.

Для защиты доступа к критичным процедурам в системе используется бит `priv` регистра E. Различаются три разновидности процедур:

- Непривилегированные процедуры, которые могут вызываться из любых процедур (как привилегированных, так и непривилегированных).
- Привилегированные процедуры, которые могут вызываться только из других привилегированных процедур.
- *Вызываемые* (*callable*) процедуры, которые могут вызываться из любых процедур, но при этом устанавливают бит `priv`. Они образуют связь между привилегированными и непривилегированными процедурами.

Различия между привилегированными, непривилегированными и вызываемыми процедурами зависят от их положения в PEP. Таким образом, системная таблица PEP (иногда называемая *SEP*) может содержать непривилегированные библиотечные процедуры. Структура таблицы показана на рис. 8.6.

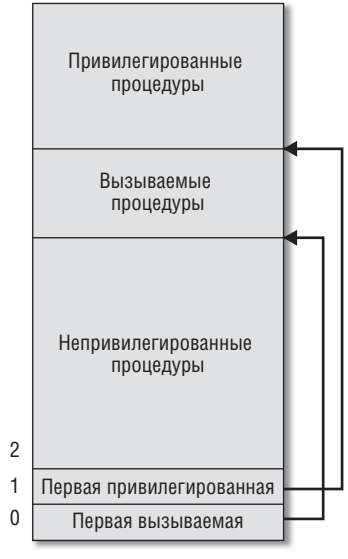


***Рис. 8.6.** Таблица точек входа в процедуры*

Действие команд PCAL и SCAL

Команда `PCAL` выполняет следующие действия:

- Если бит `priv` в регистре E не установлен (что указывает на то, что вызывающая процедура является непривилегированной), проверить значение «первая привилегированная» (слово 1 пространства кода). Если смещение в команде больше или равно этому значению, процедура пытается вызвать привилегированную процедуру. Сгенерировать системное прерывание защиты.
- Если бит `priv` в регистре E не установлен, проверить значение «первая вызываемая» (слово 0 пространства кода). Если смещение в команде больше или равно этому значению, установить бит `priv` в регистре E.
- Занести текущее значение регистра P (счетчик команд) в стек.

- Занести *старое* значение регистра E в стек.
- Занести текущее значение регистра L в стек.
- Скопировать регистр S (указатель стека) в регистр L.
- Присвоить полю RP регистра E значение 7 (пусто).
- Загрузить содержимое слова PEP, адресуемого командой в регистре P.

Команда SCAL работает точно так же, за исключением того, что она тоже устанавливает бит SC в регистре E, что гарантирует продолжение выполнения в пространстве ядра. Пространство данных не изменяется.

Команды PCAL и SCAL очень похожи, и программисту обычно не обязательно различать их. Все необходимое делает система во время выполнения. Эти библиотечные процедуры могут перемещаться между пользовательским и системным кодом без перекомпиляции.

Межпроцессорная шина

Все взаимодействия между процессорами осуществляются по *межпроцессорной шине*, или *IPB* (InterProcessor Bus). На самом деле используются две шины X и Y (см. рис. 8.1) на случай отказа одной из них. В отличие от других компонентов, обе шины используются параллельно, когда они находятся в рабочем состоянии.

Данные передаются по шине фиксированными пакетами из 16 слов. Шина работает достаточно быстро, поэтому клиентский процессор выполняет пересылку синхронно в *диспетчере* (планировщике), используя команду SEND. Процессор-получатель резервирует буфер для одной пересылки во время загрузки. При завершении пересылки получатель принимает прерывание получения по шине и обрабатывает пакет.

Ввод/вывод

Каждый процессор имеет одну шину ввода/вывода и до 32 контроллеров. Все контроллеры являются двухпортовыми и подключаются к двум разным процессорам. В любой момент времени каждый контроллер доступен только для одного процессора. Процессор, управляющий контроллером, называется его *владельцем*. Резервный канал не используется до тех пор, пока не произойдет сбой основного канала или системный оператор не переключится на него вручную.

Особенно много проблем возникает с дисками, поскольку при работе с ними возможны отказы многих компонентов. Сбой может произойти на самом диске, на физическом подключении к диску (кабель), в конт-

роллере диска, на шине ввода/вывода или на процессоре, к которому он подсоединен. В результате помимо двухпортовых контроллеров каждый диск физически дублируется (по крайней мере, теоретически), оснащается двумя портами и подключается к двум разным контроллерам, причем оба подключаются к одной и той же паре процессоров. При этом к любому контроллеру в любой момент времени может обратиться только один процессор, но один из процессоров может стать владельцем одного контроллера, в то время как другой процессор будет владельцем другого контроллера. Такая схема также желательна с точки зрения быстродействия.

На рис. 8.1 изображена типичная конфигурация: процесс ввода/вывода для системного диска `$SYSTEM` обращается к нему через процессор 0 и первый контроллер диска, в то время как процесс ввода/вывода для другого диска `$DATA`, подключенного к тем же двум контроллерам, обращается к диску через процессор 1 и второй контроллер. Процессор 0 становится владельцем первого контроллера диска, а процессор 1 – владельцем второго контроллера. Если на процессоре 0 произойдет сбой, вступит в действие резервный процесс ввода/вывода для диска `$SYSTEM` на процессоре 1, который станет владельцем контроллера, а затем продолжит выполнение. Если сбой произойдет на втором контроллере диска, то процесс ввода/вывода для `$DATA` не сможет использовать первый контроллер, поскольку его владельцем является процессор 0; процесс ввода/вывода сначала выполнит переключение, в результате чего основной процесс ввода/вывода будет работать на процессоре 0. После этого он будет обращаться к `$DATA` по тому же каналу, что и к `$SYSTEM`.

Впрочем, это все теория. На практике диски стоят дорого, и многие пользователи запускают по крайней мере часть своих дисков в минимальном режиме, без дублирования оборудования диска. Такая конфигурация работает, но, разумеется, никакой отказоустойчивости нет и в помине: фактически на одном из дисков уже произошел сбой.

Структура процессов

Guardian является микроядерной (microkernel) системой: за исключением низкоуровневых обработчиков прерываний (единственной процедуры `IOINTERRUPT`) и части очень низкоуровневого кода, весь системный сервис предоставляется системными процессами, работающими в пространствах системного кода и системных данных.

Важнейшие системные процессы:

- *Системный монитор* (PID 0 на каждом процессоре) отвечает за запуск и остановку других процессов, а также за выполнение таких разнообразных задач, как возвращение информации состояния, ге-

нерирование сообщений об ошибках оборудования и поддержание текущего времени.

- *Диспетчер памяти* (PID 1 на каждом процессоре) отвечает за ввод/вывод в системе виртуальной памяти.
- Процессы ввода/вывода, ответственные за управление устройствами ввода/вывода. Ведь доступ к устройствам ввода/вывода в системе осуществляется через специальный процесс ввода/вывода. Контроллеры ввода/вывода подключены к двум процессорам, так что каждым устройством управляет пара процессов, выполняемых на этих процессорах: *основной* процесс выполняет всю работу, а *резервный* процесс следит за состоянием основного, дожидаясь сбоя или добровольной передачи управления («переключения»).

  Главной проблемой при выборе основного процессора является текущая загрузка процессора, которую приходится регулировать вручную. Например, если между процессорами 2 и 3 подключено шесть устройств, вероятно, основные процессы для трех устройств следует разместить на процессоре 2, а основные процессы для трех других – на процессоре 3.

Парные процессы

Концепция применения парных процессов не ограничивается процессами ввода/вывода. Она является одним из краеугольных камней отказоустойчивых решений. Но чтобы понять, как работают парные процессы, необходимо понять механизм передачи сообщений в системе.

Система сообщений

Как было показано ранее, главным отличием T/16 от традиционных компьютеров было отсутствие единственных критичных компонентов. Сбой любой части системы не нарушал работоспособности всей системы. Результат больше напоминал сеть, нежели классическую мультипроцессорную машину с общей памятью.

Отказоустойчивость имела далеко идущие последствия для архитектуры операционной системы. Диск может быть подключен к любым двум из 16 процессоров. Как другие процессоры будут обращаться к нему? Современные сети для этого особого случая используют файловые системы, работающие на базе сетевых протоколов (такие как NFS и CIFS). Однако в T/16 это не особый случай, а норма.

Файловые системы – не единственная область, в которой используется такой механизм передачи данных; он также необходим и для всевозможных межпроцессных взаимодействий.

В архитектуре Tandem проблема решалась при помощи *системы сообщений*, работающей в операционной системе на очень низком уровне. Пользовательские программы не могут использовать ее напрямую.

Система сообщений обеспечивает передачу данных между процессами, и во многих отношениях напоминает более поздние протоколы TCP или UDP. Вызывающая сторона называется *инициатором*, а вызываемая – *сервером*[1].

Все коммуникации между процессами, даже выполняемыми на одном процессоре, осуществляются через систему сообщений. В их реализации используются следующие структуры данных:

- Каждое сообщение связывается с двумя блоками управления каналом связи *LCB* (Link Control Block) – и для инициатора, и для сервера. Эти маленькие объекты данных проектировались так, чтобы они поместились в один пакет IPB. Если передаваемые данные не помещаются в LCB, то к сообщению присоединяется отдельный буфер.
- Чтобы начать передачу, инициатор вызывает процедуру `link`. Процедура отправляет сообщение серверному процессу и ставит свой блок LCB в его очередь сообщений. На этой стадии серверный процесс еще не участвует в передаче, но диспетчер активизирует его при помощи события *LREQ* (Link Request).
- На стороне инициатора вызов `link` немедленно возвращает управление с информацией, необходимой для идентификации запроса; инициатору не приходится ждать, пока сервер обработает запрос.
- В какой-то момент серверный процесс получает событие `LREQ` и вызывает процедуру `listen`, которая удаляет первый блок `LCB` из очереди сообщений.
- Если с `LCB` связан буфер с данными, передаваемыми серверу, то сервер вызывает процедуру `readlink` для чтения данных.
- Затем сервер выполняет всю необходимую обработку и вызывает `writelink` для ответа на сообщение – возможно, снова с буфером данных. Это приводит к активизации инициатора по событию `LDONE`.
- Инициатор получает событие `LDONE`, проверяет результаты и завершает обмен данными вызовом `breaklink`, освобождающим связанные ресурсы.

Система сообщений используется напрямую только другими компонентами ядра; *файловая система* использует ее для взаимодействия с устройствами ввода/вывода и другими процессами. Межпроцессные взаимодействия происходят практически так же, как передача данных

[1] Эти роли по своей функциональности близко соответствуют современным терминам *клиент* и *сервер*.

при вводе/выводе, и используются также для создания отказоустойчивых *парных процессов*.

Описанный механизм по своей природе является асинхронным и многопоточным; после вызова `link` инициатор продолжает свою работу. Многие инициаторы могут отправлять запросы одному серверу, даже если последний не занимается активной обработкой запросов. Сервер не обязан немедленно реагировать на запросы `link`. Когда он отвечает, инициатор не обязан немедленно подтверждать получение ответа. В обоих случаях процесс активизируется по событию, которое обрабатывается при готовности.

Снова о парных процессах

Одно из требований отказоустойчивости заключается в том, что один сбой не должен выводить систему из строя. Мы уже видели, как эта проблема решается при вводе/выводе с помощью пар процессов; понятно, что этот метод поможет решить проблемы и с потенциальными сбоями процессоров. Таким образом, Guardian предоставляет возможность создания парных процессов пользовательского уровня.

Пара процессов состоит из *основного* и *резервного* процессов. Основной процесс выполняет основные действия, а резервный находится в состоянии «оперативного резерва». Время от времени основной процесс обновляет образ резервного процесса в памяти (*контрольную точку*). Если в основном процессе происходит сбой или он добровольно передает управление, резервный процесс продолжает выполнение с состояния последней контрольной точки. В реализации контрольных точек используется ряд процедур, вызываемых в ходе работы системы сообщений:

- Резервный процесс вызывает `checkmonitor`, чтобы перейти к ожиданию сообщений контрольных точек от основного процесса. Он остается в состоянии `checkmonitor` до тех пор, пока основной процесс не исчезнет или добровольно передаст управление. В это время процессор используется только для передачи трафика системы сообщений с целью обновления его пространства данных, а также вызовов `open` и `close` для обновления файловой информации.
- Основной процесс вызывает `checkpoint` для копирования частей пространства данных и файловой информации в резервный процесс. Программист должен решить, когда и для каких данных и файлов создавать контрольные точки.
- Основной процесс вызывает `checkopen` для создания контрольной точки открытия файла. Фактически это равнозначно вызову `open` из резервного процесса. Процесс ввода/вывода распознает резервный вызов `open` и рассматривает его как эквивалент основного вызова `open`.

- Основной процесс вызывает `checkclose` для создания контрольной точки закрытия файла. Фактически это равнозначно вызову `close` из резервного процесса.
- Основной процесс может вызвать `checkswitch`, чтобы добровольно уступить управление в паре процессов. Когда это происходит, основной и резервный процессы меняются ролями.

Когда резервный процесс возвращает управление из `checkmonitor`, он становится новым основным процессом. Управление возвращается по месту последнего вызова `checkpoint` из старого основного процесса, а не по месту вызова. Затем выполнение продолжается с этой точки.

В общем виде примерный жизненный цикл пары процессов представлен в табл. 8.1.

Таблица 8.1. Жизненный цикл пары процессов

Основной	**Резервный**
Инициализация	
Вызов `newprocess` для создания резервного процесса	
	Инициализация
	Вызов `checkmonitor` для создания контрольной точки
Вызов `checkpoint`	Ожидание в `checkmonitor`
Вызов `checkopen`	Вызов open из `checkmonitor`
Обработка	Ожидание в `checkmonitor`
Вызов `checkpoint`	Ожидание в `checkmonitor`
Обработка	Ожидание в `checkmonitor`
Добровольное переключение: вызов `checkswitch`	Получение управления
Вызов `checkmonitor` для получения данных контрольной точки	Обработка
Ожидание в `checkmonitor`	Вызов `checkpoint`
Ожидание в `checkmonitor`	Обработка
Ожидание в `checkmonitor`	Вызов `checkpoint`
Ожидание в `checkmonitor`	Сбой процессора
Получение управления	–
Обработка	

Синхронизация

Описанный механизм чрезвычайно надежен; по своей надежности он значительно превосходит классические конфигурации с жестким дублированием. В некоторых классах программных ошибок, прежде всего в *ситуациях состязаний* (*race condition*), параллельный процесс сталкивается с абсолютно такой же ошибкой, и в нем также происходит сбой. Решению с ослабленной привязкой часто удается избежать точного повторения ситуации и продолжить функционирование.

Впрочем, архитектура с парными процессами создает пару неочевидных проблем:

- Контрольные точки интенсивно расходуют ресурсы процессора. Как часто процесс должен создавать контрольные точки? Какие данные должны сохраняться в контрольной точке? Это решение принимается программистом. Если программист ошибется и забудет сохранить важные данные или сохранение будет выполнено несвоевременно, то образ памяти в резервном процессе будет искажен, что может привести к сбоям.
- Если после создания контрольной точки, но до возникновения сбоя основной процесс выполняет действия, видимые извне (например, операции ввода/вывода), то эти действия будут повторены после передачи управления резервному процессу. Это может привести к порче данных.

На практике оказалось, что некорректные контрольные точки редко создают серьезные проблемы, чего нельзя сказать о повторении ввода/вывода. Для решения этой проблемы система назначает каждому запросу ввода/вывода порядковый номер, называемый *идентификатором синхронизации*. Процесс ввода/вывода отслеживает запросы и в случае получения дублирующего запроса просто возвращает код завершения первого вызова.

Сети: EXPAND и FOX

Система сообщений T/16 фактически образует автономную сеть. Это обстоятельство упрощает работу Guardian в составе глобальных сетей: фактически система сообщений расширяется до мировых масштабов. Сетевая реализация называется *EXPAND*.

С точки зрения программиста модель EXPAND почти идеально интегрирована. Сеть может связывать до 255 систем.

Имена систем

Каждая система обладает именем, начинающимся с символа \ (например, \ESSG или \FOXII), а также номером узла. Смысл номеров узлов куда менее очевиден, чем смысл современных IP-адресов: с точки зрения программиста они необходимы разве что для кодирования имен файлов, но об этом мы поговорим чуть позже. Принципиальные различия сводятся к скорости и требованиям к доступу.

FOX

С учетом чисто практических ограничений трудно построить систему, в которой установлено более 16 процессоров; в частности, из-за аппаратных ограничений длина межпроцессорной шины не может превышать нескольких метров, так что 16 процессоров можно считать практическим максимумом. Кроме того, Tandem поддерживает быстрое оптоволоконное соединение, способное объединить до 15 систем в некое подобие локального кластера, – по сути, это высокоскоростная версия EXPAND.

Файловая система

В Tandem термин *файловая система* используется для обозначения доступа к системным ресурсам, которые могут поставлять данные («чтение») или принимать их («запись»). Помимо дисковых файлов файловая система также обслуживает устройства (терминалы, принтеры, ленточные накопители и т. д.) и процессы (межпроцессные взаимодействия).

Имена файлов

Для устройств, дисковых файлов и процессов существует общая схема назначения имен, но, к сожалению, она усложняется многочисленными исключениями. Процессы могут иметь имена, но только для процессов ввода/вывода и парных процессов имена *обязательны*. «Имя» файла всегда состоит из 24 символов, и делится на три 8-байтовых компонента. Лишь первый компонент обязателен; два других используются только для дисковых файлов и именованных процессов.

Неименованные процессы используют только первые 8 байт имени. Формат имени непарных системных процессов (например, монитора или менеджера памяти) показан на рис. 8.7.

Формат имени непарных пользовательских процессов показан на рис. 8.8.

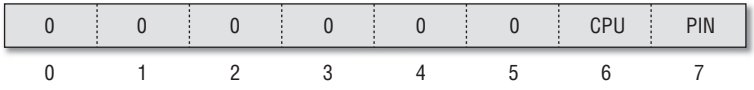


Рис. 8.7. *Формат имени непарных системных процессов*

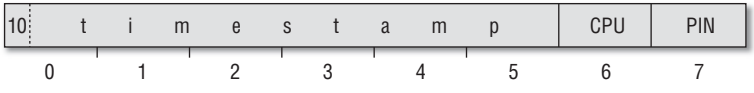


Рис. 8.8. *Формат имени непарных пользовательских процессов*

Комбинация *CPU* и *PIN* образует идентификатор процесса, или *PID* (Process ID). PIN – *идентификационный номер процесса* в пределах процессора. Таким образом, на каждом процессоре может выполняться до 256 процессов. Процессы также могут использовать два других компонента для передачи информации процессам.

Типичные имена представлены в табл. 8.2.

Таблица 8.2. Типичные имена файлов

`$TAPE`	Ленточный накопитель
`$LP`	Принтер
`$SPLS`	Процесс-спулер
`$TERM15`	Терминальное устройство
`$SYSTEM`	Системный диск
`$SYSTEM SYSTEM LOGFILE`	Системный журнал на диске `$SYSTEM`
`$SPLS #DEFAULT`	Очередь печати спулера по умолчанию
`$RECEIVE`	Очередь входных сообщений для межпроцессных взаимодействий

Если длина компонента имени меньше 8 байт, он дополняется ASCII-пробелами. Во внешнем представлении имена записываются в кодировке ASCII с разделением точками – например, `$SYSTEM.SYSTEM.LOGFILE` и `$SPLS.#DEFAULT`.

В схеме назначения имен есть и другие странности. Подымена процессов должны начинаться с символа `#`, а в именах пользовательских процессов (но не именах устройств, которые в действительности являются

именами процессов ввода/вывода) первый компонент должен завершаться кодом PID (рис. 8.9).

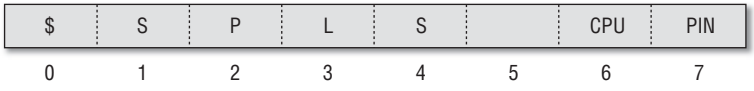


Рис. 8.9. *Формат имен именованных пользовательских процессов*

Обозначение PID в этом примере относится к PID основного процесса. Таким образом, длина имен пользовательских процессов ограничивается 6 символами, включая начальный символ $.

А если и этого недостаточно, существует отдельный набор имен для обозначения процессов, дисковых файлов и устройств в удаленных системах. В этом случае префикс $ заменяется символом \, а второй байт имени содержит номер системы, вследствие чего остаток имени сдвигается на один байт вправо. В такой схеме длина имен процессов, которые должны быть видимы в сети, ограничивается пятью символами. Таким образом, в другой системе упоминавшийся ранее процесс-спулер может иметь внешнее имя `\ESSG.$SPLS`; его внутренний формат показан на рис. 8.10.

Число 173 – номер узла системы `\ESSG`.

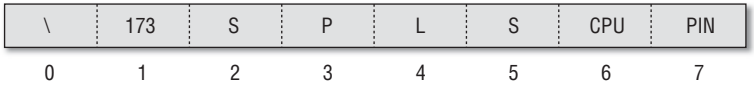


Рис. 8.10. *Формат имени процесса, видимого в сети*

Асинхронный ввод/вывод

Одной из важнейших особенностей интерфейса файловых систем является значительная ориентация на асинхронный ввод/вывод. Мы уже видели, что система сообщений асинхронна по своей природе, так что это требование реализуется относительно просто.

Процессы могут выбрать синхронный или асинхронный («неблокирующий») режим ввода/вывода в момент открытия файла. Когда файл открывается в асинхронном режиме, запрос ввода/вывода возвращает управление немедленно, сообщая только о сразу обнаруженных ошибках – например, если дескриптор файла закрыт. Позднее пользователь вызывает `awaitio` для проверки состояния запроса. Такая схема поро-

дила стиль программирования, при котором процесс выдает несколько асинхронных запросов, а потом входит в цикл вызова awaitio и обработки завершения запросов (обычно с выдачей нового запроса).

Межпроцессные взаимодействия

На уровне файловой системы межпроцессные взаимодействия предоставляют относительно прямой интерфейс к системе сообщений. При этом возникает одна проблема: система сообщений асимметрична. Инициатор отправляет сообщение и может получить ответ. Никакого аналога команды файловой системы read не существует. На другой стороне сервер читает сообщение и отвечает на него; здесь нет никакого аналога команды write.

Файловая система предоставляет процедуры read и write, но read работает только с процессами ввода/вывода. Вызовы read не работают на уровне межпроцессных взаимодействий, write на практике тоже используется нечасто. Вместо этого инициатор использует процедуру writeread, чтобы сначала передать сообщение серверу, а затем получить ответ от него. Как сообщение, так и ответ могут быть пустыми (т. е. иметь нулевую длину).

Эти сообщения попадают в очередь сообщений сервера. На уровне файловой системы очередь сообщений представляет собой псевдофайл с именем $RECEIVE. Сервер открывает $RECEIVE и обычно использует процедуру readupdate для чтения сообщения. В более поздний момент он может ответить процедурой reply.

Системные сообщения

Система использует $RECEIVE для передачи сообщений процедурам. Одним из самых важных является *стартовое сообщение*, передающее параметры только что запущенным процессам. Следующий пример написан на TAL, низкоуровневом языке системного программирования Tandem (хотя его название и означает «Tandem Application Language»). Язык TAL, созданный на основе HP SPL, напоминает Pascal и Algol. Одной из самых необычных особенностей является использование символа ^ в идентификаторах; символ подчеркивания (_) не разрешен. Приведенный пример достаточно близок к коду C; вероятно, вы разберетесь в нем без особого труда. Приведенный в нем процесс запускает дочерний серверный процесс, а затем обменивается с ним данными.

В первом фрагменте приведен родительский процесс (инициатор):

```
call newprocess (program^file^name,,,,,, process^name); -- запуск серверного
                                                        -- процесса
```

```
call open (process^name, process^fd);              -- открытие процесса
call writeread (process^fd, startup^message, 66); -- запись стартового
                                                   -- сообщения
while 1 do
  begin
  read data from terminal
  call writeread (process^fd,
                  data, data^length,     -- запись данных
                  reply, max^reply,      -- чтение данных
                  @reply^length);        -- для фактической длины ответа
  if reply^length > 0
    записать данные на терминал
  end;
```

Дочерний процесс (сервер):

```
call open (receive, receive^fd);
do
  call read (receive^fd, startup^message, 66);
until startup^message = -1;          -- первое слово стартового сообщения
                                     -- содержит -1.
while 1 do
  begin
  call readupdate (receive^fd, message, read^count, count^read);
  обработка полученного сообщения с заменой содержимого буфера
  call reply (message, reply^length);
  end;
```

Первыми сообщениями, полученными дочерним процессом, являются системные сообщения; родительский вызов `open` отправляет дочернему процессу сообщение `open`, после чего первый вызов `writeread` отправляет стартовое сообщение. Дочерний процесс обрабатывает эти сообщения и отвечает на них. Он может воспользоваться сообщением `open`, чтобы отслеживать инициаторов или получить информацию, переданную в последних 16 байтах имени файла. Только после этого процесс начинает получать нормальный трафик сообщений от родителя. На этой стадии с ним также могут взаимодействовать другие процессы. Аналогичным образом при закрытии сервера инициатором сервер получает системное сообщение `close`.

Ввод/вывод

Важно помнить, что операции ввода/вывода с устройствами, включая операции ввода/вывода с дисковыми файлами, обеспечиваются процессами ввода/вывода, так что «открытие устройства» в действительности представляет собой открытие процесса ввода/вывода. Тем не менее, в реализации ввода/вывода с устройствами и файлами существу-

ют некоторые различия, хотя в обоих случаях используется один набор процедур файловой системы. В частности, для работы с файлами обычно используются более традиционные процедуры `read` и `write`, а дисковые операции ввода/вывода выполняются в асинхронном режиме.

Безопасность

Система T/16 соответствовала стандартам своего времени и не была особо безопасной. На практике это не создавало серьезных проблем, но один момент все же заслуживает особого упоминания: переход из непривилегированных процедур в привилегированные осуществлялся на основании позиции точки входа в таблице PEP и значения бита `priv` в регистре E. Довольно рано стали очевидны уязвимости. Если кому-то удастся заставить привилегированную процедуру вернуть значение через указатель и перезаписать сохраненный в стеке регистр E с установкой бита `priv`, то процесс останется привилегированным и после возвращения из процедуры. Вызываемые процедуры сами должны проверять свои параметры-указатели и убеждаться в том, что они не приводят к исключениям адресации, а значения возвращаются только в пользовательском окружении. Ошибка в процедуре `setlooptimer`, устанавливающей сторожевой таймер с возможностью (необязательной) возврата старого значения, позволила получить привилегии `SUPER.SUPER` (привилегированный пользователь с идентификатором 255,255 или -1):

```
proc make^me^super main;
begin
int .TOS = 'S';                    -- адрес вершины стека
call setlooptimer (%2017);         -- установка таймера
call setlooptimer (0, @TOS [4]);   -- сброс, запись старого
                                   -- значения в сохраненный регистр E
pcb [mypid.<8:15>].pcbprocaid := -1; -- получение привилегий
end;
```

Второй вызов `setlooptimer` возвращает старое значение `%2017` на место сохраненного в стеке содержимого регистра E; в частности, установка бита `priv` оставляет процесс в привилегированном состоянии. Теоретически это значение можно было бы уменьшить до `%2016`, но это ничего не изменит (сохраненное поле RP не восстанавливается). Затем программа использует SG-относительную адресацию для изменения информации пользователя в своем собственном блоке управления процессом PCB (Process Control Block). Функция `mypid` возвращает PID текущего процесса, а последние 8 разрядов (`<8:15>`) содержали код PIN, используемый в качестве индекса таблицы PCB.

Конечно, ошибка была быстро исправлена, но она продемонстрировала уязвимость общего подхода: программист должен был сам проверять

параметры, передаваемые вызываемым процедурам. Подобные проблемы неоднократно встречались на протяжении жизненного цикла архитектуры.

Безопасность доступа к файлам

Подход Tandem к безопасности доступа к файлам напоминает аналогичные механизмы Unix, но пользователи могут входить только в одну группу, название которой является частью имени пользователя. Мое имя пользователя SUPPORT.GREG в числовой записи имело вид 20,102; это означало, что я принадлежу только к группе SUPPORT (20), и в этой группе мне присвоен идентификатор пользователя 102. Длина каждого поля составляет 8 бит, а полный идентификатор пользователя помещается в одном слове. Если бы я захотел стать участником другой группы, то мне понадобился бы новый идентификатор – вероятно, с другим номером (например, SUPER.GREG с идентификатором ID 255,17).

С каждым файлом ассоциируется набор битов, определяющих права доступа к файлу для владельца, группы и всех пользователей. Однако, в отличие от Unix, биты имеют другую структуру; определяются четыре вида доступа: *чтение*, *запись*, *исполнение* и *уничтожение*. Последний термин использовался в Tandem вместо *удаления*, а его необходимость объяснялась отсутствием у каталогов собственных настроек безопасности.

Для каждого из видов доступа определялся круг пользователей, которым разрешалось его использование:

- *Владелец* – только владелец файла.
- *Группа* – все участники той же группы.
- *Все* – любой пользователь в системе.

Все разрешения относятся только к той системе, в которой находился файл. Второй набор режимов доступа вводился для сетевых обращений, чтобы регулировать доступ со стороны пользователей других систем:

- *Пользователь* – только пользователь с такими же номерами пользователя и группы, как у владельца файла.
- *Класс* – любой пользователь с таким же номером группы, как у владельца файла.
- *Сеть* – любой пользователь в любой системе.

Для устройств никаких мер безопасности не существовало, и пользовательским процессам приходилось реализовывать собственные схемы. Конечно, в сетевой среде этот недостаток был особенно существенным. На семинаре по безопасности в начале 1989 года мне удалось про-

демонстрировать похищение пароля SUPER.SUPER (root) в системе \TSII, установленной посреди административной зоны в Купертино, посредством простого вывода фиктивного приглашения на системную консоль. В это время я находился в Дюссельдорфе (Германия).

Фольклор

Вспоминая прошлое из наших дней, из начала XXI века, мы часто забываем, как интересно было работать с компьютерами. Компания Tandem была интересной, и она заботилась о своих сотрудниках. Однажды в пятницу в конце 1974 года на ранней стадии разработки системы программное обеспечение наконец-то заработало на оборудовании (до этого момента все программы разрабатывались на имитаторах). Представьте, какую бурю восторга вызвало это событие. Говорят, один из вице-президентов принес ящик пива; все расселись за столом, отмечая событие и обсуждая планы на будущее. В частности, было решено, что ящик пива должен стать еженедельным мероприятием. Традиция продолжалась до 1990-х годов, когда ее признали неполиткорректной и отменили.

Фирма Tandem породила массу острот и каламбуров – кстати, одним из них было само название «Tandem». В те дни мы носили футболки с надписями типа «Две головы лучше, чем одна» и «Нас ничто не остановит». И конечно, когда кто-нибудь жаловался на избыток чего угодно, следовал стандартный ответ: «Это на случай отказа».

Последняя фраза не была просто шуткой. Она глубоко укоренилась в наших мыслях. В мае 1977 года после возвращения с пятинедельных курсов повышения квалификации Tandem меня ждало печальное известие: наша кошка сбежала. Убедившись в том, что она не собирается возвращаться, мы завели... двух новых кошек. Только много лет спустя я осознал, что это было результатом успешной психологической обработки. Даже сегодня я терпеть не могу перезагружать компьютер и делаю это только в крайних случаях.

Недостатки

Машина T/16 пользовалась выдающимся успехом в своей области (одно время свыше 80% всех банкоматов в США работало под управлением систем Tandem), но, конечно, у нее были и недостатки. Одни, например более высокая стоимость по сравнению с традиционными системами, были неизбежными. Другие были не столь очевидны для проектировщиков.

Быстродействие

Фирма Tandem по праву гордилась почти линейным масштабированием быстродействия при добавлении нового оборудования. В работе Хорста и Чу (1985), относящейся к более поздней системе TXP, показано, как кластер FOX линейно масштабируется с 2 до 32 процессоров.

Бартлетт (1982) показывает оборотную сторону: производительность системы сообщений ограничивала скорость работы даже малых систем. На передачу одного сообщения без присоединенных данных требуется около 2 мс, а передача сообщений с 2000 байт данных занимала от 4,6 мс (на одном процессоре) до 7,0 мс (на разных процессорах). Весьма значительные затраты для одной операции ввода/вывода, и даже в те дни это считалось медленным. Задержка между двумя последовательными запросами ввода/вывода к дисковому файлу была слишком большой, чтобы запросы могли быть обработаны за время прохождения головки диска над данными; таким образом, за каждый оборот диска обслуживался только один запрос. Программа, которая последовательно читала с диска и обрабатывала данные объемом 2 Кбайт (например, аналог *grep*), работала на фактической скорости передачи данных всего 120 Кбайт/с. При меньших объемах данных (скажем, 512 байт) пропускная способность обработки падала до скоростей работы с дискетами.

Аппаратные ограничения

Как подразумевает название «Tandem/16», проектировщики руководствовались 16-разрядным менталитетом. Для середины 1970-х годов это было типично, но в компьютерной среде уже зрело убеждение, что у «настоящих» компьютеров будут 32-разрядные слова. С течением времени некоторые проблемы были решены в машинах-преемниках. В 1981 году фирма Tandem выпустила систему *NonStop II* с совместимым набором команд и меньшими аппаратными ограничениями. За следующие 10 лет появились и другие совместимые, но более производительные машины. Ни одна из них не отличалась выдающейся быстротой, но им хватало производительности для обработки сетевых транзакций. Кроме того, операционная система была переписана для решения самых неотложных проблем, и с течением времени в нее вносились дополнительные усовершенствования. К их числу относились:

- Введение 31-разрядного режима адресации, предоставляющего пользовательским процессам «неограниченное» пространство памяти. В этом режиме использовались байтовые адреса, но из-за сохранения старого формата команд не снимались ограничения на размер стека и выход кода за границу 32 Кбайт.

- Повышение количества аппаратных карт виртуальной памяти. В T/16 их было всего четыре, для пространств кода и данных. Машина TNS/II, как она называлась, поддерживала 16 карт памяти; это означало, что процессор может непосредственно адресовать до 2 Мбайт без участия менеджера памяти. Одна из этих карт использовалась как буфер ассоциативной трансляции для обработки 31-разрядных расширенных адресов.
- Система Guardian II – новая версия Guardian, поставлявшаяся с TNS/II – также поддерживала пространства системной библиотеки и пользовательской библиотеки, что увеличивало общее пространство, доступное для процессов, до 384 Кбайт. Еще позднее количество библиотечных пространств было увеличено с 2 (системное и пользовательское) до 62 (по 31 для системы и пользователя) посредством переключения сегментов. В любой момент времени может быть активна только одна карта пользовательской библиотеки и одна карта системной библиотеки.
- С размером очереди сообщений возникли проблемы. Мониторные процессы регулярно отправляли сообщения состояния всем процессам, которым они были нужны. Если сообщения не читаются процессом, то большие объемы ресурсов (LCB и буферы сообщений) тратятся на хранение дубликатов сообщений. Для решения этой проблемы в Guardian II был введен процесс-*передатчик*, который хранил одну копию сообщений состояния и посылал их процессу при вызове `listen`.

Упущенные возможности

Компьютер T/16 имел революционную архитектуру, а его среда выполнения обладала возможностями, присущими лишь немногим машинам того времени. Но в конечном итоге дорогу ему преградили мелочи. Например, независимость устройств является одной из непреходящих целей для разработчиков операционных систем, и фирма Tandem прошла большую часть пути к этой цели. Однако ей все же не удалось в полной мере раскрыть весь потенциал из-за проблем с именами и другими неоправданными несовместимостями. Почему в межпроцессных взаимодействиях нельзя использовать `read`? Почему имена процессов должны отличаться по формату от имен устройств? Почему девятым байтом должен быть символ `#`?

Проблема «двуглавия»

Более серьезная проблема была обусловлена основным механизмом обнаружения ошибок. Она нормально работала при сбое только одного

компонента и обычно неплохо справлялась со сбоями двух компонентов. Но что если сбои происходили на обоих межпроцессорных шинах? Даже в двухпроцессорной системе это приводило к катастрофическим последствиям. Каждый процессор считал, что на другом процессоре произошел сбой, и пытался взять под контроль устройства ввода/вывода – причем не однократно, а постоянно. Такие ситуации встречались (к счастью, очень редко) и часто приводили к полной порче данных на дисках, совместно используемых обоими процессорами.

Последующие поколения

С началом 1990-х годов объем продаж Tandem стал снижаться. Это объяснялось целым рядом факторов:

- Компьютерное оборудование в целом становилось более надежным, что лишало продукцию Tandem конкурентных преимуществ.
- Компьютерное оборудование стало работать *намного* быстрее, выявляя изначальные ограничения быстродействия архитектуры.

В 1990-х годах процессор T/16 был заменен решением на базе MIPS, хотя большая часть архитектуры осталась неизменной. С другой стороны, даже в 2000 году в продукции Tandem процессоры MIPS использовались для эмуляции команд T/16. Одна из причин заключалась в том, что большая часть системного кода Tandem была написана на языке TAL, тесно привязанном к архитектуре T/16. Попытки перевода кодовой базы на C отвергались из-за чрезмерных затрат.

Система Tandem/16 при всей своей революционности на удивление слабо повлияла на отрасль и стандарты проектирования современных машин. Большая часть ее функциональности сейчас реализуется более доступными средствами (зеркальные диски, сетевые файловые системы, модель «клиент-сервер», возможность оперативной замены оборудования), однако мне еще не попадались решения, явно построенные по образцу Tandem. Возможно, дело в том, что архитектура T/16 сильно отличалась от большинства современных систем – и, конечно, чисто коммерческий характер среды, для которой она разрабатывалась, тоже не способствовал ее широкому распространению.

Библиография

На сайте Hewlett-Packard опубликован ряд статей, посвященных Tandem; начните с технических отчетов в разделе *http://www.hpl.hp.com/techreports/tandem/*, в частности:

Bartlett, Joel «A NonStop kernel», June 1981. *http://www.hpl.hp.com/techreports/tandem/TR-81.4.html?jumpid=reg_R1002_USEN*. (Дополнительная информация о рабочей среде операционной системы.)

Bartlett, Joel et al. «Fault tolerance in Tandem computer systems», May 1990. *http://www.hpl.hp.com/techreports/tandem/TR-90.5.html*. (Более подробное описание оборудования.)

Gray, Jim «The cost of messages», March 1988. *http://www.hpl.hp.com/techreports/tandem/TR-88.4.html*. (Описание некоторых проблем быстродействия с теоретической точки зрения.)

Horst, Robert and Tim Chou «The hardware architecture and linear expansion of Tandem nonstop systems», April 1985. *http://www.hpl.hp.com/techreports/tandem/TR-85.3.html*.

	Принципы и свойства		Структуры
	Гибкость	✓	Модуль
	Концептуальная целостность		Зависимость
	Возможность независимого изменения	✓	Обработка
✓	Автоматическое распространение		Доступ к данным
	Удобство построения		
✓	Адаптация к росту		
✓	Сопротивление энтропии		

9

JPC: эмулятор x86 PC на языке Java

Риз Ньюман
Кристофер Деннис

«Эмуляторы работают медленно, а Java – медленный язык; значит, написанный на Java эмулятор будет ползать со скоростью улитки». Как и предсказывает расхожее мнение, первый прототип JPC работал в 10 000 раз медленнее настоящего компьютера.

Тем не менее идея эмулятора x86 PC на языке Java выглядит соблазнительно – только представьте, как Linux или Windows запускается в безопасной изолированной среде («песочнице») Java... и при этом выполняется достаточно быстро для реальной работы. Задача не из простых – разработчик должен воспроизвести все функции одного из самых сложных технических устройств, созданных человеком. Чтобы повторить физическую архитектуру x86 PC на базе виртуальной машины Java и ограничить ее рамками изолированной среды аплетов Java, нам порой приходилось проявлять чудеса изобретательности.

Попутно мы столкнулись с некоторыми проблемами, которые редко встречаются современным программистам, но напоминают нам об основных принципах, обычно воспринимаемых как нечто само собой ра-

зумеющееся. В итоге у нас получилась красивая архитектура, которая доказывает, что эмуляция x86 на уровне чистого кода Java возможна и притом работает достаточно быстро для практического использования.

Введение

Рост мощности процессоров и быстродействия сетей заметен даже для рядовых пользователей. Многое из того, что несколько лет считалось нереальным, становится обыденным. Десять лет назад, когда в Калифорнии образовалась небольшая технологическая компания VMWare, сама идея создания полностью виртуального компьютера в физическом компьютере выглядела довольно странно. В конце концов, если у вас есть компьютер, зачем замедлять его за счет введения дополнительного уровня виртуализации – чтобы получить возможность запускать то, что вы и так можете запустить? Программы использовали все вычислительные мощности оборудования, а если вам требовалось увеличить объем выполняемой работы, можно было просто докупить новые компьютеры. Тогда зачем нужна виртуализация?

Прошло десять лет, и преимущества виртуальных машин уже не вызывают сомнений. Оборудование работает настолько быстро, что на современном компьютере можно запустить несколько виртуальных машин без особого ущерба для общего быстродействия, а программные службы играют настолько важную роль, что преимущества безопасности и надежности, обеспечиваемые их полной изоляцией в виртуальных машинах, стали вполне очевидными.

Тем не менее у чистой виртуализации имеются свои проблемы. Ее функционирование в той или иной степени основано на аппаратной поддержке[1]; такая близость к физической машине создает определенную неустойчивость. Напротив, эмуляторы представляют собой виртуальные компьютеры, построенные исключительно на программном уровне, а потому не предъявляющие никаких особых требований к базовому оборудованию. Эмулируемая машина полностью отделена от реальных устройств. Ее присутствие в системе не создает дополнительной нестабильности по сравнению с обычными приложениями. Запуск прикладных программ составляет основное предназначение компьютера, поэтому эмуляция всегда останется самым стабильным и надежным механизмом создания виртуальных машин.

[1] Как минимум необходимо оборудование, совпадающее с «виртуализируемым». Скажем, продукты типа Xen или VMWare позволяют создавать виртуальные x86 PC только на оборудовании x86.

Современные критики эмуляции (как и критики эмуляции десять лет назад) обращают внимание прежде всего на потери быстродействия, которые часто действительно весьма значительны. История показывает, что технологический прогресс решает проблемы скорости, но вследствие постоянного роста сложности современных аппаратных и программных средств нетривиальные взаимодействия между оборудованием, операционными системами и прикладными программами постоянно порождают новые, еще более сложные проблемы. Таким образом, задача низкоуровневого разделения систем с сохранением надежности и устойчивости при сохранении совместного доступа к физическим ресурсам становится все более насущной, но все более сложной. Эмуляция обеспечивает необходимую степень надежности, безопасности и гибкости и поэтому выглядит все более привлекательно.

В настоящее время существует немало эмуляторов; в области эмуляции x86 PC внимания заслуживают прежде всего Bochs и QEMU. Оба проекта разрабатывались в течение многих лет, оба позволяют загружать современные операционные системы и запускать прикладные программы. Однако оба проекта написаны на внутреннем коде конкретной машины (C/C++) и при переходе на новую базовую аппаратную архитектуру/стек ОС (т. е. на новый тип системы) требуют перекомпиляции. Более того, при запуске приложений с повышенной безопасностью всегда приходится учитывать возможность злонамеренной модификации эмулятора или ошибки, допускающей злонамеренное вмешательство. Например, QEMU использует динамическую двоичную трансляцию для достижения приемлемой скорости. Если в процессе трансляции обнаружится уязвимость, это приведет к нестабильной работе программ или нарушению гарантий безопасности.

Если пользователи готовы смириться с потерей скорости ради полных гарантий безопасности, почему бы не построить эмулятор на базе самой распространенной и безопасной виртуальной машины – JVM (Java Virtual Machine)? В течение 10 лет машина JVM проходила проверку на безопасность запуска кода, и пользователи часто соглашаются с запуском непроверенного кода, загруженного из Интернета, в изолированной среде Applet Sandbox – контейнере безопасности, предоставляемом JVM. В частности, JVM защищает от таких фундаментальных ошибок программирования, как выход за пределы массива и чтение данных из неинициализированной памяти. Кроме того, JVM с менеджером безопасности может заблокировать любую потенциально опасную операцию, выполняемую гостевой программой.

JPC представляет собой именно это: эмулятор x86 PC, полностью написанный на Java. JPC не содержит машинного кода; он эмулирует все стандартные аппаратные компоненты x86 PC, оставаясь исключи-

тельно в границах изолированной среды Java Applet Sandbox. Таким образом, в этих ограничениях безопасности операционные системы x86 и приложения, запускаемые в JPC, полностью изолируются от базового оборудования – до того, что оборудование может вовсе отсутствовать на x86 PC[1]. С точки зрения хоста JPC представляет собой еще одно приложение/аплет Java, поэтому хост может быть уверен в безопасности запуска кода (каким бы он ни был). JPC может загружать DOS и современные версии Linux, предоставляя гостевым программам и ОС полный доступ к оборудованию виртуальной машины, включая административные привилегии, – и все это в рамках изолированной среды Java.

Чтобы выйти за пределы JPC и выполнить потенциально опасное действие на хосте, находящемся под контролем пользователя, запустившего JVM, злоумышленник должен найти ошибку в коде JPC, которая совпала бы с уязвимостью в JVM. Таким образом, ему придется преодолеть три полностью независимых уровня защиты. Разные уровни обычно строятся разными компаниями, и поскольку они могут использоваться во множестве разных обстоятельств, их безопасность подвергается постоянному независимому тестированию и анализу. Также обратите внимание на необходимость совпадения уязвимостей при взломе. Найти ошибку в JPC, а потом перейти к задаче взлома JVM недостаточно; хакер должен найти такую ошибку JPC, которая непосредственно стыковалась бы с подходящей ошибкой безопасности в используемой машине JVM. Проект JPC распространяется с открытым кодом, поэтому анализ кода и построение «чистой» версии является относительно простой задачей; кроме того, в приложениях с высокой безопасностью может использоваться JVM с повышенной безопасностью. Таким образом, JPC ставит непреодолимую преграду на пути вредоносного кода x86 и предоставляет самую безопасную, удобную и надежную среду для изучения такого кода «в действии».

Как и в случае с виртуализацией, с ростом мощности оборудования область применения эмуляции расширяется от обычной безопасности до обеспечения отказоустойчивости в критически важных системах. Какие бы сбои ни происходили на эмулируемой машине, они никак не отражаются на работоспособности хоста, и последний может продолжить выполнение других эмулируемых экземпляров. Эмуляция предотвращает потенциальные проблемы при использовании даже тщательно

[1] Эмулятор JPC был адаптирован для запуска в среде J2ME, что позволяет ему загружать исходную немодифицированную версию DOS даже на мобильных телефонах на базе ARM11!

продуманных средств поддержки виртуализации в современном оборудовании[1].

В основной части этой главы описан процесс построения прототипа и разработки JPC. Мы также покажем, как серия последовательных усовершенствований привела к текущей версии архитектуры JPC, а затем рассмотрим дополнительные возможности и последствия применения уникальной комбинации технологий, представленной JPC.

Проверка концепции

Архитектура x86 PC существует уже более 30 лет. В своей эволюции она прошла несколько поколений оборудования. На каждой стадии особое внимание уделялось обратной совместимости, так что программа для исходного процессора 8086 с большой вероятностью запустится на современном PC. Такой подход обладает несомненными преимуществами и стал одной из причин непревзойденного успеха платформы, однако он означает, что внедрение новых технологий неизбежно приводит к повышению сложности, чтобы сохранить работоспособность существующего кода. Если бы архитектура PC строилась сегодня «с нуля», то многие ее аспекты были бы реализованы иначе – и наверняка заметно упростились бы.

И все же платформа x86 получила повсеместное распространение. Сейчас в мире насчитывается более 1 миллиарда компьютеров с архитектурой x86, причем каждый год производится еще более 200 миллионов. Следовательно, самый широкий круг пользователей будет у эмулятора, ориентированного на архитектуру x86 PC.

Однако создание такого эмулятора – задача не из простых. На программном уровне придется эмулировать такие программные компоненты, как процессор x86, жесткий диск (с контроллером), драйверы клавиатуры и мыши, видеокарта VGA, контроллер DMA, шина PCI, мост «хост-PCI», таймер, часы реального времени, контроллер прерываний и мост «PCI-ISA». У каждого устройства имеется спецификация, которую придется прочитать и преобразовать в программный код. Описание процессора x86 содержит 1500 страниц, а общий объем технических справочников составляет примерно 2000 страниц. Набор команд x86 содержит до 65 000 возможных команд, которые могут выполняться в программном коде, с четырьмя уровнями защиты страниц па-

[1] Например, аппаратная виртуализация процессоров x86 (Intel VT, AMD Pacifica) имеет дефекты безопасности, связанные с совместным использованием кэша L1/L2 на многоядерных процессорах.

мяти и четырьмя режимами работы процессора, причем команды могут работать по-разному (и работают) в зависимости от режима.

Прежде чем браться за этот титанический труд, важно оценить, оправдает ли результат потраченные усилия. Проекты Bochs и QEMU достигли поставленной цели, и изучение их подхода к решению поставленных задач даст полезную информацию. Однако эти проекты написаны на C/C++, и следовательно, польза от изучения их опыта ограничена, так как JPC придется держаться в границах среды Java со всеми вытекающими дополнительными архитектурными ограничениями и проблемами быстродействия.

Задачу можно немного упростить за счет выбора простой группы эмулируемых устройств. Так как основным «узким местом» системы станет эмуляция процессора, нет смысла эмулировать сложный, но быстродействующий контроллер жесткого диска (и сам диск). Вполне достаточно простой и надежной эмуляции жесткого диска; это относится и ко всем остальным аппаратным компонентам. Даже при выборе процессора можно остановиться на Pentium II; не обязательно эмулировать расширенные наборы команд последних моделей. Современные программы, в том числе и операционные системы, обычно прекрасно работают без этих команд, обеспечивающих обратную совместимость, так что это решение не создает сколько-нибудь существенных ограничений.

Тем не менее скорость эмуляции остается серьезной проблемой. Самым очевидным способом повышения быстродействия является использование некоторой разновидности динамической двоичной трансляции для перехода от трудоемкой пошаговой программной эмуляции к более эффективному использованию вычислительной мощности оборудования. Этот прием используется во многих разновидностях; современный процессор x86 при первом использовании разбивает команды x86 на микрокоды, кэшируемые для ускорения повторного выполнения. JIT-компилируемые среды Java аналогичным образом компилируют блоки байт-кода в машинный код, чтобы ускорить выполнение многократно повторяемых фрагментов. Эффективность подобных методов обусловлена тем фактом, что почти во всех программах 10% кода соответствуют 90% времени выполнения[1]. Простые приемы ускорения этих 10% «горячего» кода способны кардинально повысить общую скорость выполнения. Также обратите внимание на то, что их

[1] Это апокрифическое правило нашло практическое подтверждение при загрузке DOS и запуске многочисленных DOS-игр в JPC. Когда программа находится под полным контролем эмулятора, собрать подобную статистику несложно. Также см. статью Дональда Кнута «An empirical study of FORTRAN programs» (Software Practice and Experience, 1: 105-133. Wiley, 1971).

применение не требует компиляции всего кода; выборочная оптимизация может привести к значительному повышению быстродействия без неприемлемых задержек, возникающих при оптимизации всего кода.

Практически все программы (как откомпилированные из C/C++ в машинный код x86, так и существующие в виде байт-кода Java) в современных компьютерах в той или иной степени подвергаются динамической двоичной трансляции; данный факт доказывает мощь и эффективность этого приема. Таким образом, при планировании построения JPC можно надеяться, что применение аналогичных приемов позволит достичь разумной скорости.

Анализ существующих эмуляторов позволил сделать вывод о достижимости поставленной цели, а выбор простой архитектуры PC сократил исходный объем работы до приемлемого уровня. Далее мы должны были оценить потенциал динамической двоичной трансляции для повышения скорости, а также убедиться в возможности достижения реалистичного быстродействия (хотя бы теоретически). Если бы после применения всех программных ухищрений эмулятор JPC работал на скорости 1%, то за проект не стоило бы и браться.

Тестирование потенциального быстродействия процессора

Чтобы оценить потенциальный уровень быстродействия при разных тактиках эмуляции, мы придумали простую модель – процессор Toy. Процессор Toy имеет 13 команд, содержит 2 регистра и 128 байт оперативной памяти. Для проведения тестов на скорость на ассемблере Toy была написана простая программа, эквивалентная следующему фрагменту на языке C:

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 50; j++)
        memory[51 + j] += 4;
```

В исходном состоянии память заполнена нулями, а вывод программы содержит состояние памяти в конце выполнения (первые 50 байт памяти зарезервированы для программного кода).

На языке Java был написан эмулятор архитектуры Toy. 100 000 последовательных выполнений этой программы на виртуальной машине Sun HotSpot заняли 8000 мс. Выполнение на том же оборудовании программы на языке C, откомпилированной GCC (со всеми оптимизациями), заняло всего 86 мс. Наивная эмуляция привела к падению быстродействия на два порядка, что, в общем, неудивительно. Однако наша эмуляция действительно была очень простой: эмулятор каждый раз читал

следующую ассемблерную команду из памяти, выбирал операцию конструкцией `switch` и выполнял программу в цикле вплоть до завершения.

В оптимизированной версии процесс выбора и передачи управления желательно исключить. Для этого можно воспользоваться стандартным приемом подстановки (inlining); компилятор C подставляет часто используемый код на стадии компиляции, а HotSpot VM делает это во время выполнения на основании текущих показателей. После подстановки выполнение заняло 800 мс – скорость повысилась в 10 раз.

При использовании выбора с передачей управления указатель команд должен был обновляться после каждой команды, чтобы процессор знал, где следует искать следующую команду. После подстановки кода указатель обновляется только при выполнении всего подставляемого блока. Удаление лишних приращений также способствует оптимизации выполнения; HotSpot может сосредоточиться на ключевых операциях, не отвлекаясь на необходимость постоянной синхронизации указателя команд с ходом выполнения. Перемещение обновления указателя команд в конец подставляемых секций сократило время выполнения до 250 мс – дополнительный выигрыш в 3,2 раза.

Тем не менее трансляция ассемблерного кода по упрощенному автоматическому алгоритму привела ко множеству ненужных перемещений данных между памятью и регистрами. В нескольких местах результат сохранялся в байтовом массиве в памяти, после чего немедленно читался снова. Простой анализ потоков команд позволил автоматически обнаруживать и устранять дефекты такого рода. Время выполнения сократилось до 80 мс – эмулятор заработал со скоростью оптимизированного машинного кода!

Таким образом, после необходимой оптимизации времени выполнения, не отличавшейся особой сложностью, эмулятор Java смог выполнять внутренний код Toy на 100% возможной скорости оборудования. Разумеется, ожидать 100% скорости на неизмеримо более сложном оборудовании x86 PC было бы нереально, но тесты показывали, что вполне возможно добиться практически значимого быстродействия.

Таким образом, несмотря на весь начальный скептицизм по поводу концепции чистого эмулятора Java x86, эксперименты убедительно доказывали, что такая технология возможна. Далее вы узнаете, как особенности аппаратной архитектуры x86 PC отображались на JVM для обеспечения эффективной эмуляции. Также будут описаны некоторые принципиальные решения по поводу программной архитектуры, основанные исключительно на поведении JVM и способные значительно повлиять на быстродействие.

Архитектура PC

Современный PC – очень сложная машина. Его оборудование было оптимизировано и многократно совершенствовалось для построения высокоэффективной универсальной вычислительной платформы. Однако он также содержит унаследованные компоненты и функциональность, спроектированные для сохранения обратной совместимости. Задача была решена очень качественно – базовая архитектура машины IA-32 почти не изменилась с появления в 1985 году процессора 386. Более того, в отношении системной архитектуры сам процессор 386 не так уж сильно отличался от своих предшественников x86.

Хотя диаграмма на рис. 9.1 сильно упрощена, при небольших изменениях текста и дублировании некоторых блоков она может легко сойти за архитектурную диаграмму эмулятора JPC.

Основная часть JPC проектировалась на базе относительно простого системного анализа, а отображение исходной системы на JPC в основном сводилось к однозначному соответствию между аппаратными спецификациями и классами Java. Например, последовательный порт в JPC был представлен одним классом `SerialPort`, реализующим интерфейсы `HardwareComponent` и `IOPortCapable`. Архитектура, построенная на базе этого упрощенного подхода, хорошо понятна, в ней легко ориентироваться, и в целом между объектами архитектуры не существует нежелательных жестких привязок. Одним из достоинств архитектуры JPC становится гибкость – как и в реальной машине, к шине PCI можно «подключать» устройства, а замена компонентов позволяет строить виртуальные машины в широком диапазоне конфигураций.

Существует только одна причина для отказа от этой схемы: когда ясность дизайна и модульность вступают в прямое противоречие с быст-

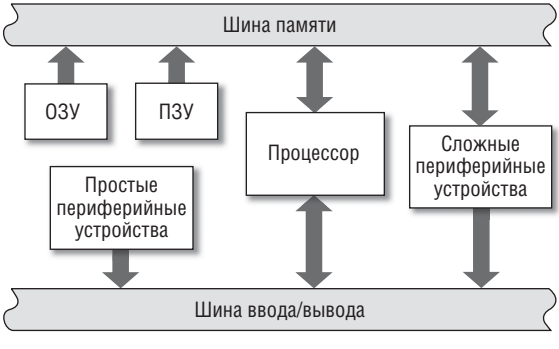


Рис. 9.1. *Базовая архитектура современного PC*

родействием. В JPC такие примеры встречаются в двух ключевых местах: при эмуляции процессора (интенсивные вычисления) и системы памяти (интенсивная передача данных). Наличие этих «особых зон» приводит к двум последствиям для проекта:

- Нарушение однородности кодовой базы. Реализация простых устройств (таких как интерфейс IDE) представляет собой простой «перевод» спецификации в программный код; она представлена всего двумя классами: `IDEChannel` и `PIIX3IDEInterface`. С намного более сложным устройством – процессором – связывается огромный объем кода. В эмуляторе он представлен восемью пакетами и более чем 50 классами.
- Разработчикам приходится менять свои ориентиры в той или иной фазе работы. Грубо говоря, в ходе работы над аппаратной эмуляцией за пределами подсистемы памяти или процессора вы стремитесь к полной ясности и модульной структуре кода. При работе над процессором или памятью главной целью становится максимальное быстродействие.

Основной защитной мерой при работе над чувствительными элементами архитектуры должен стать принцип пессимистической изобретательности. Чтобы добиться максимального быстродействия, мы должны быть постоянно готовы к экспериментам. Но даже к незначительным изменениям в кодовой базе следует относиться с подозрением – пока не будет доказано, что они, по крайней мере, не оказывают отрицательного влияния на быстродействие.

Именно требование максимального быстродействия в «узких местах» эмуляции сделало работу над проектом JPC такой интересной. В оставшейся части этой главы мы расскажем, как нам удалось построить архитектуру логичную и удобную в сопровождении – но без ущерба для быстродействия системы.

Быстродействие в Java

Первое правило оптимизации: Не делай этого.

Второе правило оптимизации (только для экспертов): Подожди с оптимизацией.

Майкл А. Джексон

Следующие советы, как и любые рекомендации из области быстродействия, не являются непреложными истинами. Хорошо спроектированный и четко запрограммированный код почти всегда намного предпоч-

тительнее «оптимизированного» кода. Применяйте эти рекомендации только тогда, когда это приносит очевидный положительный эффект или эта «последняя капля» быстродействия действительно необходима.

Совет №1: избегайте создания лишних объектов

Лишние экземпляры объектов (особенно с коротким сроком жизни) ухудшают быстродействие. Это объясняется тем, что размножение объектов приводит к частой уборке мусора в младших поколениях объектов, а эти алгоритмы обычно действуют по принципу «я работаю, все ждут».

Совет №2: используйте статические методы

Если метод можно сделать статическим, так и поступите. Статические методы не являются виртуальными, поэтому динамическая диспетчеризация к ним не применяется. Современные виртуальные машины могут применять подстановку для статических методов намного проще и чаще, чем для методов экземпляров.

Совет №3: выбор по таблице – хорошо, выбор с поиском – плохо

Команды выбора `switch`, метки которых образуют относительно компактное множество, работают быстрее, чем команды с разнородными метками. Дело в том, что в Java предусмотрено два байт-кода для конструкций выбора: `tableswitch` и `lookupswitch`. Табличный выбор реализуется с использованием косвенных вызовов, а значение метки определяет смещение в таблице функций. Выбор с поиском происходит намного медленнее, потому что для него необходимо найти подходящую пару *значение:функция*.

Совет №4: отдавайте предпочтение компактным методам

Маленькие фрагменты кода предпочтительны, так как JIT-среды обычно рассматривают код на уровне методов. Большой метод с «интенсивной» областью может быть откомпилирован полностью. Число непопаданий в кодовый кэш увеличивается, а это плохо отражается на быстродействии.

Совет №5: исключения должны быть исключительными

Исключения должны инициироваться в исключительных ситуациях, а не при возникновении обычных ошибок. Применение исключений для передачи управления в особых ситуациях дает возможность VM оптимизировать «не-исключительный» путь с достижением оптимального быстродействия.

Совет №6: осмотрительно применяйте паттерн «декоратор»

Паттерн «декоратор» хорошо смотрится с архитектурной точки зрения, но дополнительный уровень абстракции может обойтись слиш-

ком дорого. Помните, что декораторы можно не только добавлять, но и удалять. Удаление может рассматриваться как исключительная ситуация, а для его реализации может использоваться специализированное исключение.

Совет № 7: instanceof для классов работает быстрее

Вызов `instanceof` для класса выполняется значительно быстрее, чем для интерфейса. Модель простого (одиночного) наследования Java означает, что для класса `instanceof` реализуется одним вычитанием и одной выборкой по массиву; для интерфейса вызов `instanceof` реализуется поиском по массиву.

Совет № 8: сведите к минимуму использование synchronized

Ограничьтесь минимальным размером блоков `synchronized`; они могут привести к лишним затратам. Постарайтесь заменить их атомарными операциями или `volatile`-ссылками.

Совет № 9: остерегайтесь внешних библиотек

Избегайте использования внешних библиотек, возможности которых превышают ваши прямые потребности. Если задача проста и критична, постарайтесь закодировать ее во внутренней реализации; скорее всего, специализированное решение лучше подойдет для вашей задачи, обеспечит лучшее быстродействие и сократит количество внешних зависимостей.

Накладные расходы

Многие проблемы, с которыми мы сталкивались в ходе разработки и проектирования JPC, были связаны с накладными расходами. При попытке эмуляции полноценной вычислительной среды только одно не позволяет вам использовать фактическое быстродействие базового оборудования на 100%: затраты ресурсов на эмуляцию. Часть этих затрат имеют временную природу (например, эмуляция процессора), другие – пространственную (затраты памяти).

Самым очевидным примером проблем, возникающих из-за пространственных затрат, является адресное пространство: 4-гигабайтное (32-разрядное) пространство памяти виртуального компьютера не поместится в 4-гигабайтном (или меньшем) пространстве реального оборудования хоста. Даже при больших объемах памяти хоста мы не можем использовать объявления вида `byte[] memory = new byte[4 * 1024 * 1024 * 1024];`. Эмулируемое адресное пространство необходимо каким-то образом сократить до пространства одного процесса на хостовом компьютере, а в идеале – оставить побольше свободного места!

Чтобы сэкономить память, мы сначала отметим, что адресное пространство 4 Гбайт практически никогда не заполняется до отказа. На типичном компьютере обычно установлено до 2 Гбайт памяти, а в большинстве случаев удается обойтись значительно меньшей величиной. А поскольку не все 4-гигабайтное пространство будет занято физической памятью, максимальные требования быстро сокращаются.

Проектирование эмулируемого физического адресного пространства начинается с небольшого экскурса в будущее. Одна из функций блока управления памятью IA-32 поможет нам в формировании структуры адресного пространства. В защищенном режиме блок управления памятью процессора делит адресное пространство на 4-килобайтные фрагменты, называемые *страницами*. Следовательно, в нашем эмуляторе естественно разделить память на блоки того же размера.

Деление адресного пространства на фрагменты по 4 Кбайт означает, что данные уже не сохраняются непосредственно в адресном пространстве. Вместо этого данные хранятся в атомарных блоках памяти, представленных различными подклассами, производными от `Memory`. В адресном пространстве сохраняются ссылки на эти объекты. Итоговая структура и схема обращений к памяти показаны на рис. 9.2.

Примечание

Чтобы оптимизировать вызовы `instanceof`, мы спроектировали цепочку наследования объектов `Memory` без использования интерфейсов.

Структура состоит из 2^{20} блоков; для хранения содержимого каждого блока требуется 32-разрядная ссылка. Сохранение этих ссылок в массиве (наиболее очевидное решение) приводит к дополнительным затратам памяти в 4 Мбайт, что в большинстве случаев несущественно.

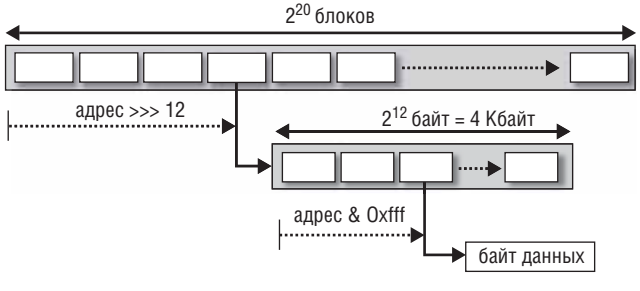


Рис. 9.2. *Структура блоков физического адресного пространства*

Совет №7: instanceof для классов работает быстрее

Вызов `instanceof` для класса выполняется значительно быстрее, чем для интерфейса. Модель простого (одиночного) наследования Java означает, что для класса вызов `instanceof` реализуется одним вычитанием и одной выборкой по массиву; а для интерфейса он реализуется поиском по массиву.

Если эти затраты памяти неприемлемы, можно выполнить дополнительную оптимизацию. Память в физическом адресном пространстве делится на три категории:

ОЗУ

Физическая память отображается с нулевого адреса и следует в направлении возрастания адресов. Для нее характерны частые обращения с малой задержкой.

ПЗУ

Микросхемы ПЗУ могут размещаться по любому адресу. Обращения к ПЗУ относительно редки, но задержка также мала.

Ввод/вывод

Области отображаемого на память ввода/вывода могут размещаться по любому адресу. Обращения к ним происходят достаточно часто, но обычно сопровождаются большей задержкой, чем обращения к ОЗУ.

Для адресов, принадлежащих диапазону адресов ОЗУ реального компьютера, используется однофазное преобразование, гарантирующее минимальную задержку при обращениях к ОЗУ. Для обращений к другим адресам, занимаемым микросхемами ПЗУ и областями ввода/вывода, отображаемыми на память, используется двухфазное преобразование (рис. 9.3).

Таким образом, чтение данных из ОЗУ выполняется за три этапа:

```
return addressSpace.get(address);
    return blocks[i >>> 12].get(address & 0xfff);
        return memory[address];
```

А чтение из более старших адресов выполняется в четыре этапа:

```
return addressSpace.get(address);
    return blocks[i >>> 22][(i >>> 12) & 0x3ff].get(address & 0xfff);
        return memory[address];
```

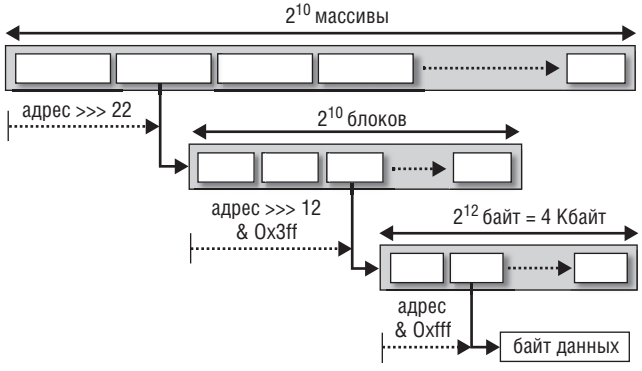


Рис. 9.3. *Физическое адресное пространство с двухфазным преобразованием*

Двухуровневая оптимизация экономит память, одновременно предотвращая создание «узких мест» при обращениях к ОЗУ. Каждый вызов и уровень абстракции при чтении из памяти выполняет свою функцию. Именно так и должна использоваться абстракция – не ради определения четких интерфейсов, а для достижения идеального баланса быстродействия и затрат памяти.

Если существует вероятность того, что выделенная память не будет использована, в JPC всегда применяется отложенная инициализация. Таким образом, адресное пространство нового экземпляра JPC содержит физическое адресное пространство с привязками к объектам `Memory`, не занимающими памяти. Когда 4-килобайтная секция ОЗУ в первый раз используется при чтении или записи, объект полностью инициализируется, как показано в листинге 9.1.

Листинг 9.1. Отложенная инициализация

```
public byte getByte(int offset)
{
    try {
        return buffer[offset];
    } catch (NullPointerException e) {
        buffer = new byte[size];
        return buffer[offset];
    }
}
```

Опасности защищенного режима

Появление защищенного режима вводит в игру совершенно новую систему управления памятью, в которой поверх физического адресного пространства появляется дополнительная прослойка. В защищенном режиме может быть включено страничное преобразование, обеспечивающее возможность перестановки 4-килобайтных блоков физического адресного пространства. Для управления перестановкой используется серия таблиц, находящихся в памяти; содержимое таблиц может динамически изменяться кодом, выполняемым на машине. На рис. 9.4 показана полная схема страничного преобразования.

Теоретически каждое обращение к памяти на машине потребует полного поиска по структуре страниц для определения физического адреса, соответствующего заданному линейному адресу. Так как этот процесс весьма сложный и затратный, реальная машина кэширует результат в буферах ассоциативной трансляции (TLB). Помимо дополнительных уровней косвенной адресации страничные преобразования сопряжены с дополнительными мерами безопасности. С каждой страницей может быть связан пользователь или супервизор, а также статус чтения или чтения/записи. При попытке кода обратиться к страницам без наличия необходимых привилегий (а также при обращениях

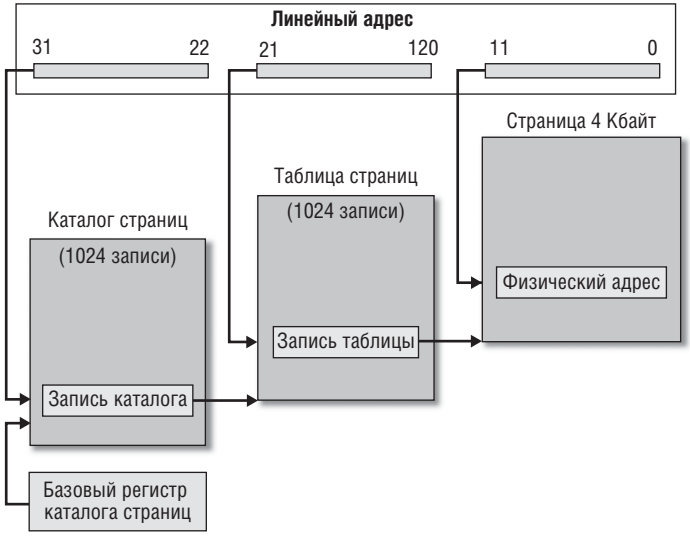


Рис. 9.4. *Механизм страничного преобразования в архитектуре IA32 (только для страниц 4 Кбайт)*

к несуществующим страницам) процессор инициирует исключение по аналогии с программными прерываниями.

Понятно, что оптимизация такой структуры должна начинаться с имитации реального процессора; иначе говоря, мы должны реализовать некую форму кэширования табличных преобразований. Начнем с констатации двух ключевых фактов:

- Отображение страниц выполняется блоками по 4 Кбайт. Конечно, такая величина блока была выбрана для нашего физического адресного пространства не случайно.
- Когда процесс защищенного режима обращается к памяти для чтения и записи, он просто видит результат отображения на память этих 4-килобайтных блоков (притом что обращения к некоторым блокам могут привести к исключениям процессора). Физическое адресное пространство представляет собой один из возможных порядков исходных объектов `Memory` (в котором все объекты расположены в порядке следования адресов), но в целом этот порядок ничуть не важнее любого другого порядка.

Из сказанного можно сделать вывод, что самой естественной формой кэша (т. е. наших таблиц TLB) является дублирование структуры физического адресного пространства. Страницы памяти отображаются в соответствии с новым порядком, определяемым страничными таблицами При первом обращении к адресу, принадлежащему некоторой странице, выполняется полный перебор табличной структуры для нахождения объекта памяти физического адресного пространства. Ссылка на объект помещается в соответствующую позицию линейного адресного пространства, и в дальнейшем остается в кэше до тех пор, пока мы не решим удалить ее.

Для решения проблем доступа (чтение/запись, пользователь/супервизор) было принято тактическое решение, которое повышает скорость ценой дополнительных затрат памяти. Мы создаем линейное адресное пространство для каждой возможной комбинации: чтение-пользователь, чтение-супервизор, запись-пользователь, запись-супервизор. Операции чтения и записи памяти используют соответствующие индексы. Переключение между пользовательским режимом и режимом супервизора сводится к изменению всего двух ссылок в системе памяти, как показано на рис. 9.5.

Примечание

Такая схема довольно близко соответствует реализации страничной трансляции в ядре Linux. В системе Linux в линейное адресное пространство каждого процесса пользовательского режима отображаются страницы режима ядра. На аппаратном уровне это предотвращает переключение контекста для передачи

управления кода ядра с целью выполнения служебных операций или системных функций. Наши переключения в пространство ядра сводятся к простому переключению ссылок на массивы, требующему минимальных затрат ресурсов.

Остается решить проблемы с обращениями к отсутствующим страницам, или *страничными отказами* (*page faults*) и нарушениями защиты. Такой подход к обработке этих ситуаций в определенной степени подтверждает наше понимание истинной природы исключений как в низкоуровневом процессорном смысле, так и в языке Java. Его можно сформулировать следующим образом:

`Exception` – исключение, а `Error` – ошибка.

А если подробнее, объект `Exception` должен представлять редкие или исключительные, но не обязательно фатальные ситуации; объект `Error` должен представлять фатальные или однозначно нежелательные ситуации.

Учитывая широкие возможности обработки исключений в языке Java, несколько странно, что многие программисты неохотно пользуются механизмом обработки исключений в своей работе. Механизм исключений представляет самый эффективный и чистый способ обработки редких, но исправимых ситуаций. Инициирование исключений поможет эффективно оптимизировать стандартный путь выполнения, но увеличит затраты, связанные с обработкой неизбежных исключительных ситуаций. Из нашего обсуждения очевидно следует, что страничные отказы и нарушения защиты являются исключительными, но нормальными событиями, которые должны представляться экземплярами `Exception`.

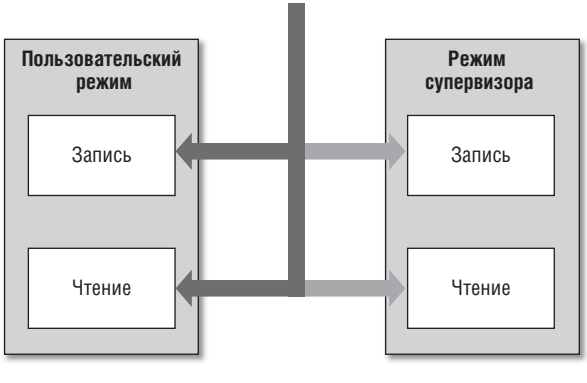


Рис. 9.5. *Переключение уровней защиты в линейном адресном пространстве*

Совет №5: исключения должны быть исключительными

Исключения должны инициироваться в исключительных ситуациях, а не при возникновении обычных ошибок. Применение исключений для передачи управления в особых ситуациях дает возможность VM оптимизировать «не-исключительный» путь с достижением оптимального быстродействия.

При обработке страничных отказов и исключений мы вполне сознательно проигнорировали два стандартных правила использования исключений:

- Класс `ProcessorException`, представляющий все типы исключений процессора Intel, расширяет класс `RuntimeException`.
- Страничные отказы, а в меньшей степени и нарушения защиты, являются исключительными ситуациями; тем не менее они встречаются достаточно часто, чтобы для сохранения быстродействия мы инициировали их в виде статических экземпляров исключений.

Для обоих решений существовали веские причины, но оба были приняты в основном из-за специфических особенностей проекта JPC.

Решение о расширении `RuntimeException` в основном было принято по эстетическим соображениям. Традиционно класс `RuntimeException` был зарезервирован для исключений, которые не могут быть обработаны программой (и не должны обрабатываться). В большинстве книг рекомендуется пропускать их наверх, чтобы исключение привело к завершению программного потока. Как известно, различия между проверяемыми исключениями и исключениями времени выполнения относятся к «синтаксическим удобствам» из той же категории, что и параметризация, автоупаковка примитивов и циклы `for-each`. Мы вовсе не пытаемся очернить эти конструкции; просто если классификация исключений не влияет на откомпилированный код, мы вполне можем выбрать самую удобную классификацию без какого-либо риска для производительности. В нашем случае исключение `ProcessorException` оформлено как исключение времени выполнения, чтобы нам не приходилось включать объявление `throws ProcessorException` в многочисленные методы.

Выдача статистически инициализированных экземпляров исключений (то есть по сути синглетных исключений) для страничных отказов и нарушений доступа приносит пользу на обоих концах цепочки обра-

ботки исключений. Во-первых, мы избавляемся от затрат на создание экземпляра нового объекта каждый раз, когда потребуется выдать исключение. Тем самым предотвращаются затраты на реконструкцию трассировки стека вызывающего потока (в современной JVM задача отнюдь не тривиальная). Во-вторых, экономятся затраты на определение природы типа исключения, так как с ограниченным набором статических исключений, определяющих тип, достаточно провести серию ссылочных сравнений.

Безнадежное дело

Нигде в архитектуре IA-32 ее популярность не проявляется так наглядно, как в наборе команд. Простая архитектура на базе простых аккумуляторных регистров эпохи 8080 за прошедшие годы превратилась в обширное, сложное семейство команд. Процессор IA-32 стал RISC-подобной микросхемой с многочисленными дополнениями и немыслимым количеством режимов адресации.

Если смотреть на происходящее с точки зрения Java-программиста, возникает соблазн начать писать классы, словно чем больше степень структурирования кода, тем проще становится задача. Такое решение неплохо (если не идеально) сработало бы при разработке дизассемблера. Однако в системе, в которой создаются многочисленные объекты, неизбежно происходит интенсивная уборка мусора. Это приводит к двойным потерям: не только от затрат на создание множества объектов, но и от частых уборок мусора. В современных средах уборки мусора с делением на поколения (примером которой является Sun JVM) небольшие объекты с малым сроком жизни создаются в молодом поколении, а почти все алгоритмы уборки мусора в молодом поколении работают по принципу «я работаю, все ждут». Таким образом, декодер, создающий многочисленные объекты, будет работать медленно не только из-за неоправданного создания объектов, но и ввиду многочисленных (пусть и очень коротких) пауз уборки мусора.

По этой причине очень важно свести к минимуму создание объектов в декодере, управляющем интерпретацией. В декодере реального режима этот минималистский подход приводит к созданию классов из 6500 строк, содержащих всего 42 вхождения ключевого слова `new`:

- 4 × `new boolean[]` во время загрузки класса (`static final`)
- 3 × `new Operation()` для циклического буфера
- 2 × `new int[]` для расширяемого буфера в `Operation`
- 33 × `new IllegalStateException()` на разных путях выполнения

После создания экземпляра декодера конструирование объектов необходимо только для расширения буферов `int[]` в `Operation`. После расширения буферов конструирование объектов и уборка мусора отсутствуют, а следовательно, декодирование происходит без лишних пауз.

Архитектура декодера демонстрирует то, что мы считаем одним из важнейших канонов программирования (в нашем случае – на Java):

«Можно» не значит «нужно».

В нашем случае JVM может выполнять автоматизированную уборку мусора, но это не значит, что ее обязательно использовать. В коде, критичном по быстродействию, следует с осторожностью подходить к созданию объектов. Остерегайтесь незаметного создания объектов в таких классах, как `Iterator` или `String`, или при вызовах `varargs`.

Совет №1: избегайте создания лишних объектов

Лишние экземпляры объектов (особенно с коротким сроком жизни) ухудшают быстродействие. Это объясняется тем, что размножение объектов приводит к частой уборке мусора в младших поколениях объектов, а эти алгоритмы обычно действуют по принципу «я работаю, все ждут».

Микрокод: когда «меньше» означает «больше», и наоборот

Итак, мы создали декодер для разбора потока команд IA-32, работающий без уборки мусора, но еще не разобрались, в какую форму должны декодироваться команды. Архитектура IA-32 не относится к системам команд с фиксированной длиной; длина команд лежит в диапазоне от 1 байта до 15 байт (максимум). Сложность набора команд в основном обусловлена различными режимами адресации, которые могут использоваться для любого операнда команды.

На исходном верхнем уровне каждая операция делится на четыре фазы:

Входные операнды

Загрузка данных операции из регистров или памяти.

Операция

Обработка входных операндов.

Выходные операнды

Сохранение результатов операции в регистрах или памяти.

Операции с флагами

Изменение битов регистра флагов для описания результата операции.

Такое разбиение позволяет отделить простую операцию от сложности операндов. Скажем, операция `add eax,[es:ecx*4+ebx+8]` изначально разбивается на пять операций:

```
load eax
load [es:ecx*4+ebx+8]
add
store eax
updateflags
```

Понятно, что `load [es:ecx*4+ebx+8]` тоже не является простой операцией, и ее можно легко разбить на несколько меньших элементов. Даже для одного этого формата адресации существуют:

- Шесть возможных сегментов памяти.
- Восемь возможных индексных регистров.
- Восемь возможных базовых регистров.

Таким образом, только для подмножества режимов адресации существуют 384 возможные комбинации. Очевидно, к вычислению адреса необходимо применить дополнительное разбиение. Обращения к памяти продолжают делиться до тех пор, пока мы не получаем:

```
load eax
memoryreset
load segment es
inc address ecx*4
inc address ebx
inc address imm 8
load [segment:address]
add
store eax
updateflags
```

Чтобы создать эмулированный набор команд с приемлемым быстродействием, нужно найти баланс между двумя группами приоритетов:

- Во-первых, время декодирования необходимо сбалансировать со временем выполнения, чтобы оптимизировать общую скорость. Следует помнить, что в интерпретируемом процессоре главной целью является исходное выполнение с минимальной задержкой. Код «нормальной» ветви выполнения должен добраться до более позд-

них стадий оптимизации. Наша цель здесь – получить на выходе код без блокировки всей эмуляции. Оптимизации на поздних стадиях могут осуществляться асинхронно, а время, затраченное на ранних стадиях, блокирует выполнение. Итак, нам нужен относительно простой набор быстро декодируемых команд.

- Во-вторых, размер набора команд необходимо сбалансировать с длиной «откомпилированного» кода. Естественно, компактный набор команд генерирует более объемистый код, а с большим набором команд код должен получиться более компактным. Мы говорим «должен», потому что и в большом, но неудачно выбранном наборе команд, могут потребоваться длинные секции. Интерпретатор меньшего набора занимает меньше памяти и содержит меньше кода; соответственно каждая из его операций будет выполняться намного быстрее, но ему придется выполнить более широкий набор команд. Таким образом, мы ищем разумный баланс между размером набора и длиной кода, чтобы добиться от интерпретатора почти оптимальной производительности.

При поиске оптимальной точки по обоим показателям важно постоянно помнить Правило Хоара:

Преждевременная оптимизация –
корень всех зол в программировании.

Тони Хоар[1]

Для многих оптимизаций идеальная точка баланса зависит от системы. В среде Java система включает не только физическое оборудование, но и JVM. При наличии дополнительного фактора обязательной JIT-компиляции Java-компонентов среды маломасштабные тесты быстродействия крайне ненадежны. Соответственно мы старались воздерживаться от принятия решений на основе тестов и доверяли тестам только при выявлении действительно значительных сдвигов в быстродействии. В JIT-компилируемой среде небольшие изменения в «микротестах» в лучшем случае не воспроизводятся в другой системе. В худшем случае они так сильно зависят от сценария тестирования, что не воспроизводятся сколько-нибудь надежно даже в той же системе.

Примечание

Важная особенность набора микрокода состоит в том, что целые значения, присваиваемые константам, образуют последовательность. В интерпретаторе

[1] *http://en.wikiquote.org/wiki/C._A._R._Hoare*

центральное место занимает команда `switch` для выбора константы по набору; соответственно мы должны позаботиться о том, чтобы команда `switch` выполнялась как можно быстрее.

Учитывая все сказанное, после нескольких экспериментов мы пришли к выводу, что набор, содержащий около 750 кодов для полной эмуляции целочисленных и вещественных операций, обеспечивает хороший баланс поставленных целей. Таким образом, преобразование операций x86 в микрокоды приводит примерно к 10-кратному увеличению микрокода. На первый взгляд это очень много, но микрокод быстро декодируется, а операции получаются в разумной степени атомарными, что делает их хорошими кандидатами для последующей оптимизации.

Совет №3: выбор по таблице – хорошо, выбор с поиском – плохо

Команды выбора `switch`, метки которых образуют относительно компактное множество, работают быстрее, чем команды с разнородными метками. Дело в том, что в Java предусмотрено два байт-кода для конструкций выбора: `tableswitch` и `lookupswitch`. Табличный выбор реализуется с использованием косвенных вызовов, а значение метки определяет смещение в таблице функций. Выбор с поиском происходит намного медленнее, потому что для него необходимо найти подходящую пару *значение:функция*.

Берем JVM под контроль

«Java-программы работают медленно» – это расхожее мнение преследует Java-программистов и в наши дни. Оно главным образом базируется на опыте программистов, работающих на других языках и столкнувшихся с ранними версиями JVM во второй половине 1990-х годов. Всем нам, кто работает с Java, отлично известно, что виртуальные машины Java стремительно развивались. Движущая сила этих усовершенствований также стала ключом к ускорению JPC: окружение традиционных Java-процессов очень просто делится между областями программ и областями данных.

Из рис. 9.6 видно, что область данных дополнительно делится на статические данные (которые могут быть известны во время компиляции) и динамические данные (заведомо неизвестные). Большую часть статических данных в среде Java составляют загружаемые байт-коды

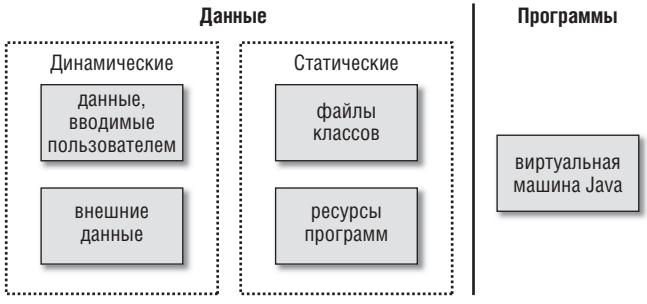


Рис. 9.6. *Области программ и данных в процессе Java*

классов. Хотя классы загружаются как данные, понятно, что в действительности они представляют код и будут интерпретироваться JVM во время выполнения. Очевидно, нам хотелось бы каким-то образом переместить байт-коды классов на другую сторону диаграммы.

В JIT-компилируемых средах, таких как Sun HotSpot, часто используемые секции кода транслируются, или динамически компилируются в машинный набор команд хоста. При этом байт-код класса перемещается из области данных в область программ. Далее код классов выполняется как низкоуровневый код, что ускоряет выполнение программы до естественной скорости оборудования (рис. 9.7).

В JPC мы воспользовались тем обстоятельством, что не все данные классов должны быть известны при запуске JVM. В терминологии Java «статические данные» правильнее было бы называть «финальными данными» (final data). При загрузке класса его байт-код фиксиру-

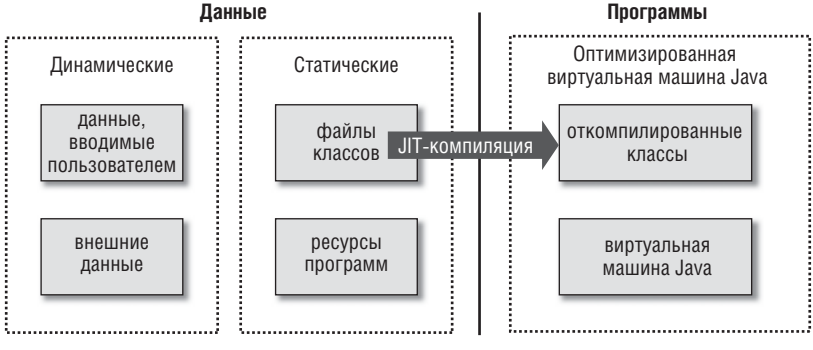


Рис. 9.7. *JIT-компиляция в среде Java*

ется, и его дальнейшее изменение невозможно (для удобства мы пока забудем о JVM TI[1]). Это позволяет нам определять новые классы во время выполнения – эта концепция хорошо знакома всем, кто работает с модульными (plug-in) архитектурами, аплетами и веб-контейнерами J2EE.

Трюк JIT-компиляции, выполняемой JVM, повторяется на уровне JPC. Таким образом, в JPC появляется второй уровень деления информации «программы/данные» в рамках JRE (Java Runtime Environment). Таким образом, процесс компиляции делится на две фазы:

1. Машинный код IA-32 компилируется в байт-код в JPC. Блоки x86 становятся действительными файлами классов Java, которые JVM может загружать как таковые.
2. JVM компилирует классы в машинный код хоста. Поскольку JVM не отличает исходные «статические» классы с написанным вручную кодом от динамических классов, построенных автоматически, оба типа оптимизируются для обеспечения максимально возможного быстродействия.

После этих двух фаз компиляции исходный код IA-32 преобразуется в архитектуру хостовой машины (рис. 9.8). Если повезет, количество команд в новом коде окажется ненамного больше исходного, а это означает, что по быстродействию сгенерированный код будет не сильно уступать оригиналу.

Итак, теперь мы знаем, как повысить скорость эмуляции, однако в предшествующем описании мы обошли подробности реализации. Эти

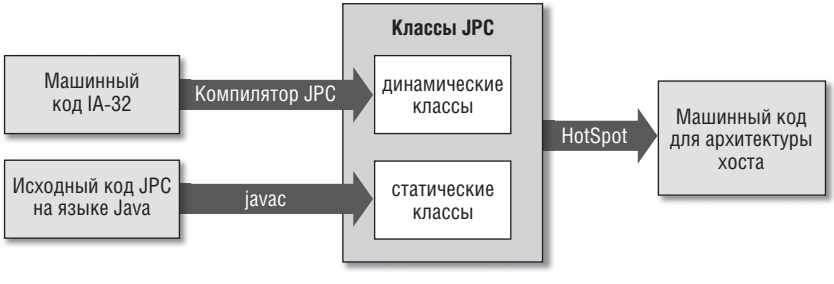


Рис. 9.8. *Три компилятора в архитектуре JPC*

[1] Тем, кто любит повозиться с JVM на низком уровне, Tool Interface (*http://java.sun.com/javase/6/docs/technotes/guides/jvmti/*) предоставляет очень интересные возможности, включая переопределение файлов классов.

подробности (и наши новые проблемы) распадаются на две категории: как выполнить компиляцию и как загружать полученные классы.

Компиляция: как изобрести велосипед

Компилятор, описанный здесь, не является самым удачным примером, однако мы находимся в слегка необычной ситуации. И *javac*, и компилятор JPC являются компиляторами первой ступени; это означает, что они просто подают свой вывод второй ступени, будь то интерпретатор байт-кода или JIT-компилятор. Как известно, *javac* прилагает минимум усилий к оптимизации своего вывода; выходной байт-код *javac* представляет собой результат обычной трансляции входного кода Java (листинг 9.2).

Листинг 9.2. Неоптимальная компиляция javac

```
public int function()
{
    boolean a = true;
    if (a)
        return 1;
    else
        return 0;
}
```

```
public int function();
    0:   iconst_1
    1:   istore_1
    2:   iload_1
    3:   ifeq    8
    6:   iconst_1
    7:   ireturn
    8:   iconst_0
    9:   ireturn
```

Минимальная оптимизация полностью оправдана, потому что большая часть работы по оптимизации выполняется на последующих стадиях компиляции. В JPC, наряду с теми же причинами, также действует фактор максимальной скорости:

- Компиляция должна осуществляться с минимальными затратами, чтобы не отнимать вычислительных мощностей от других потоков эмуляции.
- Компиляция должна осуществляться с минимальной задержкой, чтобы интерпретируемый класс заменялся как можно быстрее. Компилятор с высокой задержкой может обнаружить, что к моменту завершения компиляции надобность в коде уже отпала.

Простое генерирование кода

Задача компиляции в JPC сводится к простой трансляции микрокодов одного интерпретируемого базового блока в набор байт-кодов Java. В первом приближении мы будем считать, что базовый блок не может выдавать исключения. Из этого следует, что блок является строго определенным, имеет ровно одну точку входа и одну точку выхода. Каждая переменная, изменяемая базовым блоком, может быть выражена

как функция набора входных регистров и состояния памяти. Во внутренней реализации JPC эти функции совокупно представлены в виде однонаправленного ациклического графа.

Источник графа

Источники представляют входные данные в форме значений регистров или непосредственных данных команд.

Сток графа

Стоки представляют выходные данные в форме значений регистров или операций записи в порты ввода/вывода. Каждой переменной состояния, изменяемой блоком, соответствует один приемник.

Ребро графа

Ребра представляют пути распространения значений переменных в графе.

Узел графа

Узлы представляют операции, выполняемые с входящими ребрами; результаты операций распространяются по выходящим ребрам. В JPC операции представляют отдельный модифицирующий компонент интерпретируемого микрокода. Таким образом, один микрокод может отображаться на несколько узлов графа, если он затрагивает несколько переменных.

Преобразование графа интерпретируемых базовых блоков в байт-код Java осуществляется простым перебором графа в глубину от каждого стока. Каждый узел графа получает верхние элементы стека в качестве входных данных и оставляет результат на вершине, готовый к обработке дочерними узлами (рис. 9.9).

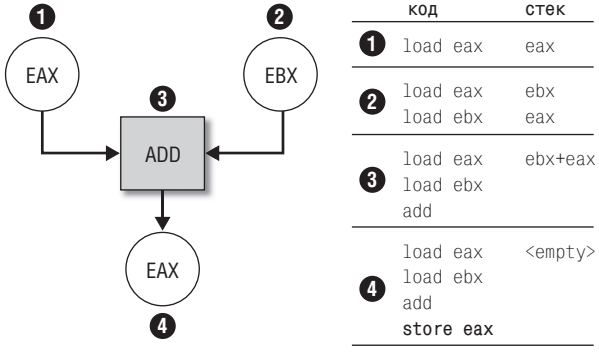


Рис. 9.9. *Представление операций x86 в виде направленного ациклического графа*

Примечание

Классический оптимальный порядок перебора графа требует, чтобы в каждом узле предки рассматривались по порядку глубины. Узел с самым длинным путем к самому дальнему источнику рассматривается первым, а узел с самым коротким путем рассматривается последним. Для регистровых операций данный метод приводит к построению самого короткого кода, потому что он устраняет большую часть жонглирования с содержимым регистров. На стековых машинах (к числу которых относится JVM) аналогичный процесс принятия решений приводит к коду с минимальной глубиной стека. В JPC мы пренебрегаем подобными приятными мелочами и доверяемся JVM – конечный результат просто не оправдывает всех сложностей с отслеживанием глубин узлов.

Разбор в порядке «от стока к стоку, по убыванию глубины» обеспечивает естественную оптимизацию графа (рис. 9.10). Изолированные секции графа, соответствующие неиспользуемому коду, недоступны при переборе от стоков, поэтому они будут автоматически удалены, так как разбор до них никогда не дойдет. Повторно используемые секции кода обрабатываются несколько раз за один разбор графа. Результат их обработки можно кэшировать в локальной переменной и загружать его при последующих переходах к узлу, чтобы избежать дублирования кода.

Код, связанный с узлом дерева, представляется одной статической функцией, откомпилированной по источнику из кодовой базы JPC. Сгенерированный код представляет собой серию операций занесения

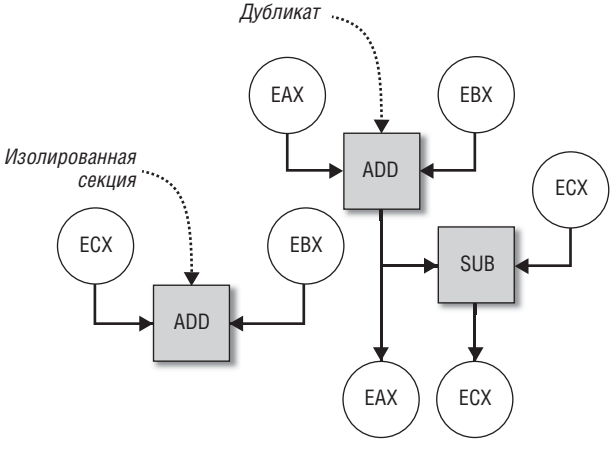


***Рис. 9.10.** Особенности направленного ациклического графа*

процессорных переменных и непосредственных данных в стек, за которыми следует серия вызовов `invokestatic` для каждого узла и, наконец, серия извлечений значений в объект процессора.

Совет №2: используйте статические методы

Если метод можно сделать статическим, так и поступите. Статические методы не являются виртуальными, поэтому динамическая диспетчеризация к ним не применяется. Современные виртуальные машины могут применять подстановку для статических методов намного проще и чаще, чем для методов экземпляров.

Обработка исключений

Итак, мы разобрались с компиляцией базовых блоков, и теперь нам придется подпортить радужную картину. Как упоминалось выше, исключения не всегда являются ошибками. Страничные отказы и нарушения защиты сплошь и рядом инициируются вполне естественным образом. При выдаче исключения состояние процессора должно соответствовать последнему успешному выполнению операции. Очевидно, это требование влечет весьма серьезные последствия для компилятора. Так как исключения IA-32 отображаются на исключения Java, мы знаем, что у проблемы есть только одно реальное решение: перехватить исключение и, оказавшись внутри обработчика, удостовериться в том, что состояние соответствует последней успешной операции.

Любой путь выдачи исключения в конкретном базовом блоке обладает теми же особенностями, что и сам базовый блок. Путь имеет одну точку входа и одну точку выхода; от основной ветви выполнения отличается только местонахождение точки выхода. А раз путь исключения почти не отличается от самого базового блока, естественнее всего использовать для его представления отдельный направленный ациклический граф. Граф исключения будет использовать тот же набор узлов, что и базовый блок, но с другим набором стоков, соответствующих его собственной точке выхода.

Код обработчика исключений строится перебором графа выбранных путей. Узлы, общие с главным путем, в момент выдачи исключения возвращаются к своему состоянию в точке перебора основного пути. Это означает, что все кэшированные и вычисленные значения основного пути могут повторно использоваться в обработчике события, избавляя нас от выполнения повторной работы.

Манипуляции с байт-кодом

Преобразование откомпилированных секций байт-кода в загружаемые классы – задача, для решения которой существует целый ряд проверенных, хорошо проработанных решений. Библиотека Apache BCEL (Byte Code Engineering Library), по словам ее создателей, помогает удобно «анализировать и создавать (двоичные) файлы классов Java, а также выполнять с ними различные операции[1]». Или ASM – «универсальная инфраструктура для анализа и манипуляций с байт-кодом Java[2]».

К сожалению, нам нужно всего лишь изменить один метод (всегда один и тот же) в одном классе-скелете. Генерироваться может только небольшое подмножество возможного набора байт-кодов, и никакой анализ нам не нужен. Похоже, что и BCEL, и ASM выходят за рамки необходимости. Например, наш алгоритм глубины стека адаптирован для быстрой оценки максимальной глубины стека в наших методах. Хотя этот алгоритм не подходит для компиляции обобщенных классов, для наших целей его вполне достаточно, и он более эффективен.

> **Совет №9: остерегайтесь внешних библиотек**
>
> Избегайте использования внешних библиотек, возможности которых превышают ваши непосредственные потребности. Если задача проста и критична, постарайтесь закодировать ее во внутренней реализации; скорее всего, специализированное решение лучше подойдет для вашей задачи, обеспечит более высокое быстродействие и сократит количество внешних зависимостей.

Загрузка и выгрузка классов в больших масштабах

Итак, у нас есть классы, и их необходимо загрузить. Первый и наиболее очевидный вопрос: «Сколько?» Рис. 9.11 дает представление о масштабах проблемы. Загрузка 100 000 классов на одной машине JVM создает, мягко говоря, определенные проблемы.

Итак, нам придется найти место для всех этих классов. JVM хранит файлы классов в специальной долгосрочной области памяти, называемой «Permanent Generation», за пределами стандартной кучи (heap)

---

[1] *http://jakarta.apache.org/bcel*

[2] *http://asm.objectweb.org/*

объектов. В типичной Sun JVM эта область начинается с 16 Мбайт и может расти до 64 Мбайт максимум. Разумеется, 100 000 классов не поместятся в 64-мегабайтную кучу. У проблемы есть два решения.

Первое решение – команда *java* `-XX:MaxPermSize=128m`. Мы просто увеличиваем размер долгосрочной области. При всей своей примитивности эта команда помогает. К сожалению, это решение имеет временный характер, потому что мы всего лишь откладываем неизбежное. Рано или поздно загруженные классы заполнят новое пространство, а расширять область до бесконечности невозможно.

Второе решение основано на сокращении количества загруженных классов. Разве уборщик мусора не должен уничтожить все неиспользуемые классы? В том, что касается уборки мусора, классы не отличаются от объектов в куче. Класс может быть уничтожен уборщиком мусора (выгружен) только при отсутствии «живых» ссылок на этот класс. Конечно, каждый экземпляр класса содержит сильную ссылку на объект `Class` своего типа. Таким образом, чтобы класс был уничтожен, в системе не должно остаться живых экземпляров этого класса, а также дополнительных ссылок на класс. Возможно, в решении проблемы нам поможет определение собственного загрузчика классов (листинг 9.3).

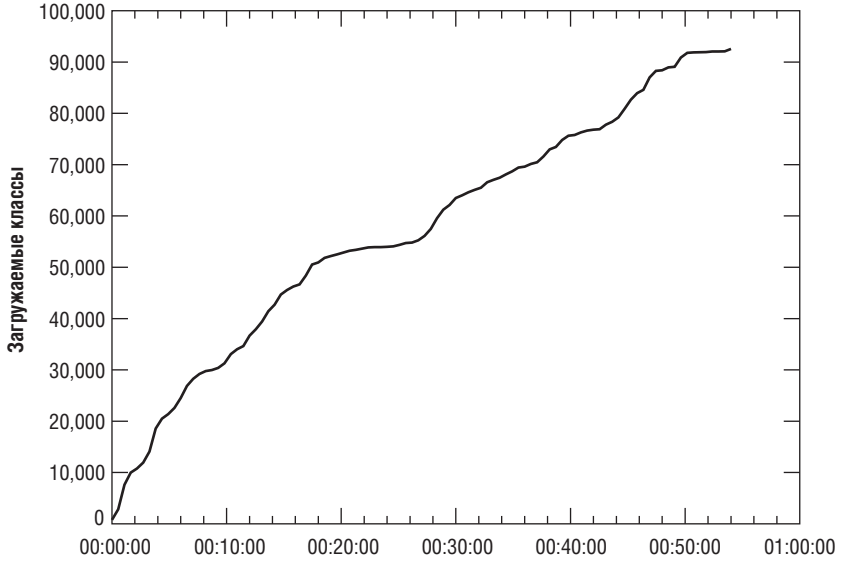


Рис. 9.11. *Количество классов в современной загрузке GNU/Linux*

Листинг 9.3. Простой загрузчик классов с ускоренным освобождением

```
public class CustomClassLoader extends Classloader
{
    public Class createClass(String name, byte[] classBytes)
    {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

    @Override
    protected Class findClass(String name) throws ClassNotFoundException
    {
        throw new ClassNotFoundException(name);
    }
}
```

Загрузчик из листинга 9.3 не сохраняет ссылки на определяемые им классы. Каждый класс представлен синглетным экземпляром без отношений, `findClass` может вполне безопасно инициировать исключение `ClassNotFoundException`. Когда синглетный экземпляр каждого класса становится кандидатом на уборку мусора, сам класс переходит в такое же состояние, и оба могут быть уничтожены уборщиком. Вроде бы все должно работать, все классы успешно загружаются. Однако возникает неожиданная проблема. По какой-то неизвестной причине классы не выгружаются – никогда. Все выглядит так, словно в системе существует объект, хранящий ссылки на наши классы.

Давайте присмотримся к списку вызовов при определении нового класса:

```
java.lang.ClassLoader: defineClass(...)
java.lang.ClassLoader: defineClass1(...)
ClassLoader.c: Java_java_lang_ClassLoader_defineClass1(...)
vm/prims/jvm.cpp: JVM_DefineClassWithSource(...)
vm/prims/jvm.cpp: jvm_define_class_common(...)
vm/memory/systemDictionary.cpp: SystemDictionary::resolve_from_stream(...)
vm/memory/systemDictionary.cpp: SystemDictionary::define_instance_class(...)
```

Что делать, если ответ «так, и все тут» нас не устраивает? Погрузившись в недра JVM, мы обнаруживаем следующий фрагмент кода:

```
// Регистрация только что загруженного класса (включение в Vector).
// Обратите внимание: регистрация выполняется до обновления
// словаря, так как возможна ошибка OutOfMemoryError
// (в этом случае мы *не* помещаем класс в словарь
// и не обновляем иерархию классов).
if (k->class_loader() != NULL) {
  methodHandle m(THREAD, Universe::loader_addClass_method());
  JavaValue result(T_VOID);
```

```
  JavaCallArguments args(class_loader_h);
  args.push_oop(Handle(THREAD, k->java_mirror()));
  JavaCalls::call(&result, m, &args, CHECK);
}
```

который обращается с вызовом на уровень Java к экземпляру загрузчика, выполняющего загрузку класса:

```
// Классы, загруженные данным загрузчиком.
// Единственное назначение таблицы - предотвратить уничтожение
// классов уборщиком мусора до уничтожения самого загрузчика.
private Vector classes = new Vector();

// Вызывается ВМ для регистрации каждого класса,
// загруженного данным загрузчиком.
void addClass(Class c) {
    classes.addElement(c);
}
```

Теперь мы знаем, какой объект хранит ссылки на все наши классы, мешая их уничтожению. К сожалению, поделать с этим ничего нельзя. Вернее, почти ничего без нарушения одной из абсолютных истин и объявления класса в пакете *java.lang*. Мы знаем, что суперкласс будет «помогать» нам с хранением ссылок. Что это означает в контексте выгрузки класса?

Из рис. 9.12 видно, что до тех пор, пока все экземпляры всех классов, загруженных загрузчиком, не станут кандидатами на уничтожение, все классы так и останутся загруженными. Таким образом, один ак-

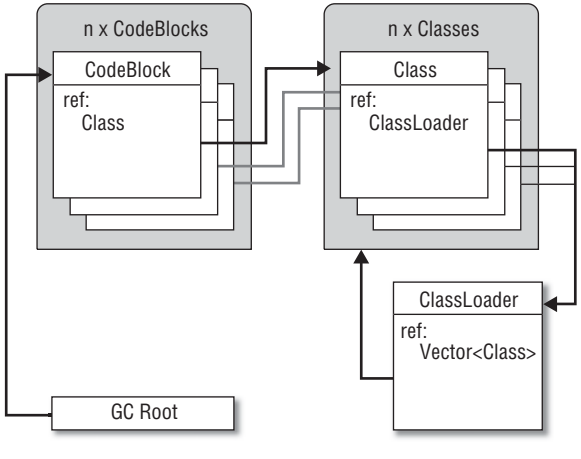


Рис. 9.12. *Схема уборки мусора для классов*

тивный CodeBlock предотвращает выгрузку *n* классов. Конечно, это нежелательно.

У проблемы существует простое решение. Никто не сказал, что значение *n* должно бесконтрольно расти. Если мы ограничим количество классов, загружаемых одним загрузчиком, то мы тем самым снизим вероятность того, что класс окажется «в заложниках». В JPC используется нестандартный загрузчик, который по умолчанию загружает не более 10 классов. Загрузка 10 класса инициирует конструирование нового загрузчика, который будет использоваться для следующих 10 классов, и т. д. В этом случае один класс может удерживать до 10 «заложников» (листинг 9.4).

Пример 9.4. Реализация загрузчика классов в JPC

```
private static void newClassLoader()
{
    currentClassLoader = new CustomClassLoader();
}

private static class CustomClassLoader extends ClassLoader
{
    private int classesCount;

    public CustomClassLoader()
    {
        super(CustomClassLoader.class.getClassLoader());
    }

    public Class createClass(String name, byte[] b)
    {
```

Кодовый кэш HotSpot

В JPC при запуске сложных заданий, требующих большого количества классов, возникает еще одно ограничение памяти. В виртуальных машинах Sun HotSpot JIT-компилированный код хранится в специальной области, называемой кодовым кэшем. JPC не только создает большое количество классов, но и обладает необычным свойством: относительно высокая доля этих классов является кандидатами для HotSpot. Это означает, что кэши HotSpot будут быстро заполняться. Таким образом, любое расширение области долгосрочного хранения классов также должно сопровождаться расширением кодового кэша.

```
            if (++classesCount == CLASSES_PER_LOADER)
                newClassLoader();

            return defineClass(name, b, 0, b.length);
        }

        protected Class findClass(String name) throws ClassNotFoundException
        {
            throw new ClassNotFoundException(name);
        }
    }
```

Замена CodeBlock

Итак, у нас имеется откомпилированный, загруженный экземпляр кодового блока. Теперь этот блок необходимо каким-то образом переместить туда, где он должен находиться. Механизм перемещения тесно связан с механизмом изначального планирования блоков для выполнения (листинг 9.5).

Листинг 9.5. Паттерн «Декоратор» для планирования компиляции

```
public class CodeBlockDecorator implements CodeBlock
{
    private CodeBlock target;

    public int execute()
    {
        makeSchedulingDecision();
        target.execute();
    }

    public void replaceTarget(CodeBlock replacement)
    {
        target = replacement;
    }
}
```

Из листинга 9.5 видно, что декоратор `CodeBlock` перехватывает вызовы и принимает решение о том, нужно ли ставить блок в очередь на компиляцию. Кроме того, существует метод, способный заменить «цель» декоратора другим экземпляром блока. После того как блок будет откомпилирован, декоратор становится бесполезным, так что в идеале нам хотелось бы заменить его. Замена реализуется очень просто. Заменяя исходный интерпретированный блок другим блоком, приведенным в листинге 9.6, мы можем передать оповещение о новом блоке вверх по стеку вызовов. Когда исключение достигнет нужного уровня, ссылка на декоратора заменяется прямой ссылкой на откомпилированный блок.

Листинг 9.6. Замена для CodeBlock

```
public class CodeBlockReplacer implements CodeBlock
{
    private CodeBlock target;

    public int execute()
    {
        throw new CodeBlockReplacementException(target);
    }

    static class CodeBlockReplacementException extends RuntimeException
    {
        private CodeBlock replacement;

        public CodeBlockReplacementException(CodeBlock compiled)
        {
            replacement = compiled;
        }

        public CodeBlock getReplacementBlock()
        {
            return replacement;
        }
    }
}
```

Совет №6: осмотрительно применяйте паттерн «декоратор»

Паттерн «декоратор» хорошо смотрится с архитектурной точки зрения, но дополнительный уровень абстракции может обойтись слишком дорого. Помните, что декораторы можно не только добавлять, но и удалять. Удаление может рассматриваться как исключительная ситуация, а для его реализации может использоваться специализированное исключение.

Максимальная гибкость

Взяв на вооружение все описанные приемы, мы получаем высокооптимизированную систему эмуляции, которая может расширяться и совершенствоваться без существенной архитектурной переработки. К системе можно подключить улучшенный компилятор; остальные компоненты тоже могут адаптироваться и заменяться другими реализациями с различными целями. Несколько примеров:

- Данные, образующие виртуальный жесткий диск, могут предоставляться (по требованию) произвольным сервером, находящимся в другой стране.
- Взаимодействие с пользователем эмулируемой системы (виртуальный экран, клавиатура и мышь) может осуществляться с удаленной системы.
- JPC может запускать программы x86 на любых стандартных виртуальных машинах Java 2; следовательно, базовое оборудование может выбираться независимо от выбора операционной системы и программного обеспечения. Кроме того, вы можете сохранить полное состояние виртуальной машины, «замораживая» ход ее работы. Затем выполнение продолжается в более поздний момент или на другом физическом компьютере, при этом программы и не подозревают о прерывании работы.

Гибкость мобильного доступа к данным

JPC позволяет разместить образ диска вместе с полным кодом JPC и JVM на флэш-накопителе. Вы подключаете накопитель к любому компьютеру, «загружаете» компьютер, читаете электронную почту и выполняете другую работу. После завершения и отключения накопителя на хосте не остается никаких следов вашего пребывания.

Возможен и другой вариант: образ жесткого диска хранится на сервере в Интернете, а вы обращаетесь к своему собственному компьютеру из другого места. Для этого достаточно загрузить локальную копию JPC и ввести ссылку на ваш сервер. В сочетании с аутентификацией и безопасностью транспорта такое решение становится мощным инструментом для организации мобильной работы. JPC, естественно, работает в среде Java, а это означает, что в качестве терминала удаленного доступа может использоваться практически любое устройство, от броузера с поддержкой Java до мобильных устройств.

При выполнении конфиденциальной работы в среде с высокой степенью защиты требования безопасности и целостности данных могут обеспечиваться на аппаратном уровне – для этого вы работаете с локальными экземплярами JPC, жесткие диски которых находятся на защищенном сервере. Каждый пользователь полностью контролирует виртуальное оборудование, на котором он работает, что повышает эффективность его труда. С другой стороны, вывести данные за пределы системы не удастся даже при взломе рабочего компьютера: локальная машина практически ничего не знает о приложении, выполняемом в гостевой операционной системе, работающей в JPC, запущенном в JVM, и т. д.

Даже если работники пользуются полным доверием, использование виртуальных машин (особенно таких гибких, как JPC) позволяет в любой момент заменить физическое оборудование. Таким образом, в области резервного копирования и восстановления после сбоев, с сохранением полного состояния всей машины (а не только данных на жестком диске), эмуляторы типа JPC обладают рядом важных преимуществ в ситуациях, когда важно мгновенное преодоление сбоев (даже при доступе через глобальную сеть).

Вследствие аппаратной независимости JPC эти сценарии в равной степени применимы к оборудованию, не относящемуся к платформе x86; такое оборудование идеально подходит для тонких клиентов, а пользователи получают доступ к привычной им среде x86.

Гибкость аудита и поддержки

Экран, клавиатура и мышь работающего экземпляра JPC могут отслеживаться и полностью контролироваться из удаленной системы, наделенной соответствующими полномочиями. Например, удаленная запись нажатий клавиш и снимков экранов при выполнении подозрительных операций поможет собрать доказательства для борьбы с компьютерными мошенниками.

Так как JPC работает с оборудованием на низком уровне, эта функция не может быть отключена техническим специалистом, который попытается обнаружить и удалить программы-наблюдатели из эмулируемой операционной системы. Даже пользователь с правами администратора (в гостевой системе) не сможет укрыться от наблюдения, каким бы компетентным он ни был.

Система аудита JPC просто сохраняет информацию о выполняемых операциях, сканируя действия в реальном времени, или идет дальше и запрещает выполнение некоторых действий. Например, при содействии серверных программ отслеживаемый экземпляр может просканировать выходные видеоданные в поиске ненадлежащих изображений и скрыть их (или заменить другими изображениями) на уровне виртуальной видеокарты. Такой низкоуровневый мониторинг означает, что пользователь не сможет обойти систему защиты контента посредством установки альтернативных программ просмотра.

Эффективность удаленной поддержки значительно повысится, если служба поддержки будет точно видеть, что происходит на всем экране, и напрямую работать с мышью и клавиатурой на уровне виртуального оборудования. Причем данная возможность сохраняется даже при запуске в JPC операционных систем, в которых удаленный доступ вооб-

ще не реализован, например системы DOS, которая продолжает использоваться во многих отраслях и странах по всему миру.

Гибкость мобильных вычислений

Вместо того чтобы запускать основную эмуляцию на локальном ресурсе и импортировать данные с удаленных ресурсов по мере необходимости, можно запустить основную эмуляцию на центральном сервере JPC. Так как для работы JPC нужна только стандартная машина JVM, центральный сервер JPC может работать на оборудовании, полностью отличном от обычного x86 PC. Уже существует несколько кандидатов, способных решать эту задачу; работа JPC была продемонстрирована на 96-ядерном вычислительном комплексе Azul. Также можно отметить серверы Sun на базе Niagara и системы, построенные с использованием технологии мобильных телефонов (эмулятор JPC уже использовался для загрузки DOS на Nokia N95, системе на базе ARM11).

Но зачем использовать централизованный сервер для запуска экземпляров JPC? Предполагается, что любой ресурс в Интернете сможет запустить экземпляр JPC по поручению кого-то другого, а экранный вывод и пользовательский ввод будут передаваться по сети владельцу виртуальной машины. При такой организации работы окружающий мир рассматривается как «N» пользователей с «M» компьютерами; между пользователями первой группы и машинами второй группы не существует фиксированных отношений принадлежности. Если машина свободна, любой пользователь может воспользоваться ею, удаленно запустив экземпляр JPC для работы с образом своего диска. Если свободная машина вдруг срочно понадобится для других целей, экземпляр JPC «замораживается», а его состояние перемещается на другой свободный физический ресурс.

Такой режим вряд ли подойдет пользователям интерактивных приложений, для которых приостановка и возобновление работы по Интернету займут слишком много времени. Однако он выглядит вполне логично для пользователей, которым требуется параллельно запустить несколько виртуальных машин без их особого участия. Обычно в таком режиме работают пользователи, применяющие крупные пакетные комплексы для выполнения параллельных задач: построения кадров анимационного фильма, поиска лекарств посредством имитации на молекулярном уровне, оптимизации инженерных архитектур и расчета сложных финансовых инструментов.

Максимальная безопасность

Запуск непроверенного кода на компьютере сопряжен с риском, и этот риск постоянно увеличивается. В Интернете постоянно появляются новые разновидности вредоносных программ – «троянов», «клавиатурных шпионов», «спам-ботов» и «вирусов». Если не принимать меры предосторожности при запуске программ, загруженных из неизвестного или непроверенного источника, возникает риск потери данных, хищения персональной информации, мошенничества или, хуже всего, – соучастия в киберпреступлении.

Кажется, на месте каждой бреши в безопасности, исправленной производителями популярных операционных систем и броузеров, возникают две новых. Как же тогда вообще можно выполнить код, который действительно расширит возможности просмотра или реализует новые полезные функции?

Выполнение кода Java в изолированной среде Java Applet Sandbox уже больше десяти лет предоставляет необходимый уровень безопасности. Добавьте дополнительную независимую прослойку безопасности, представляемую JPC, и у вас появляется среда с двойной изоляцией для выполнения небезопасного кода. Сайт JPC (*http://www-jpc.physics.ox.ac.uk*) демонстрирует, как JPC может загружать DOS и запускать классические игры в виде стандартных аплетов веб-страницы; другими словами, он показывает, что небезопасный код x86 (DOS) может выполняться в полностью защищенном контейнере на любом компьютере.

У выполнения JPC в изолированной среде имеется один серьезный недостаток: ограничения безопасности не позволяют JPC создавать загрузчики классов, поэтому динамическая компиляция, в немалой степени ускоряющая работу JPC, становится невозможной. К счастью, благодаря гибкости, заложенной в архитектуру JPC, это ограничение можно обойти без ущерба для безопасности.

Код Java в изолированной среде может загружать классы из сети – при условии, что они находятся на том же сервере, на котором изначально находился код аплета. Мы воспользовались этим обстоятельством и построили удаленный компилятор, который компилирует классы по требованию экземпляров JPC, выполняемых в аплетах, и отправляет их этим экземплярам по запросу машины JVM, ответственной за их запуск.

Локальная машина JVM рассматривает эти классы как статические ресурсы, которые просто подгружаются по требованию (в отличие от других классов JPC), тогда как фактически эти классы компилируются по мере надобности экземплярами аплетов JPC.

Такое решение обеспечивает нам скорость компилируемых классов даже в рамках среды Java Applet Sandbox. С другой стороны, пользователи могут быть уверены: какой бы код ни выполнялся в JPC, машина JVM соблюдает основные ограничения, которые позволяют безопасно выполнять код в нашем небезопасном мире.

Полезный побочный эффект удаленной компиляции заключается в том, что с течением времени он начинает использоваться многими экземплярами JPC; так появляется библиотека откомпилированных классов для совместного использования. Сервер компиляции быстро превращается в простой веб-сервер, поставляющий ранее откомпилированные классы. Более того, подсчет количества запросов к каждому откомпилированному классу позволяет серверу определить, какие классы используются наиболее интенсивно, а следовательно, оптимизации какого кода следует уделить больше внимания. Хотя последняя возможность еще не реализована, мы полагаем, что целенаправленная оптимизация может скомпенсировать снижение скорости клиентских аплетов Java, обусловленное сетевой задержкой при загрузке классов. Таким образом, клиентские аплеты Java по быстродействию могут сравняться с клиентскими приложениями JPC с локальной компиляцией[1].

Переработка архитектуры

Всем известно, что со второй попытки
все получается лучше.

Генри Форд «Моя жизнь, мои достижения»

В ходе разработки в академической среде возникают свои трудности, несколько отличающиеся от трудностей работы в коммерческих условиях. В научных средах требования к быстродействию в основном устанавливаются добровольно; это имеет как положительные, так и отрицательные последствия. Разработчики должны действовать дисциплинированно, чтобы удерживать проект «на ходу» и не допускать смещения целей. Однако некоммерческая среда также способствует быстрой разработке и проверке идей с целью подтверждения или опровержения их преимуществ. Атмосфера «свободного поиска» чрезвычайно важна для успеха самых творческих и амбициозных проектов.

[1] Если сервер компиляции связан с аплетом JPC по нормальной 100-мегабитной локальной сети, то сетевые задержки практически незаметны. Кроме того, удаленная компиляция освобождает локальные ресурсы, и это обстоятельство дополнительно компенсирует сетевые задержки.

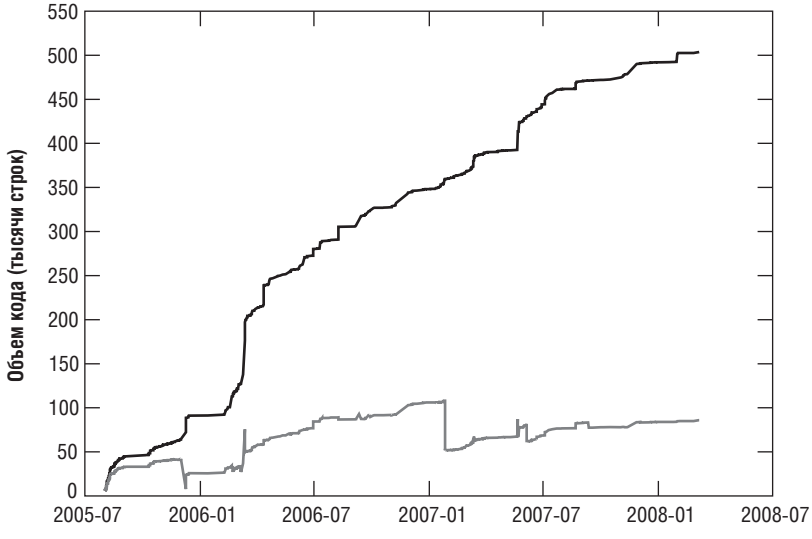


Рис. 9.13. *Накопление кодовой базы в проекте JPC*

Тот факт, что архитектура JPC развилась до текущего состояния при столь малочисленной группе разработчиков[1], объясняется общим подходом к программированию. Как видно из рис. 9.13, в ходе жизненного цикла проекта было написано свыше 500 000 строк кода. Из них до сегодняшнего дня дожило всего 85 000. Код эмуляции многократно переписывался; в частности, код один раз был полностью переписан «с нуля».

Обеспечить непрерывный цикл переработки и совершенствования кода довольно трудно, хотя в академической среде эта задача решается проще, чем в коммерческих средах. Если вы не испытываете особой эмоциональной привязанности к своему коду, то процесс удаления кода будет благотворным – не только для кодовой базы, но и для отношения к ней разработчика. Генри Форд был прав: вторая попытка почти всегда оказывается более успешной. Если серия итеративных улучшений имеет четкое условие завершения, все будет хорошо.

Итак, путь к «красивой архитектуре» состоит из четырех этапов:

[1] Средний размер группы во время основного 30-месячного цикла разработки составлял всего 2,5 программиста, причем не все участники были специалистами по Java.

1. Рассмотрите большую, сложную проблему в совокупности. Разбейте ее на несколько логичных и простых этапов, которые позволят построить полноценный прототип системы. Каждый этап представляет систему, более простую и менее функциональную, чем конечная цель, но при этом каждый этап может тестироваться в контексте своих архитектурных ограничений как часть прототипа всей системы, а не как отдельный прототип некоторого подмножества общей архитектуры (как при более традиционном модульном тестировании).
2. Прежде чем строить очередную часть очередного этапа, четко представьте себе, какой аспект вы разрабатываете и зачем. В идеале на каждом этапе должны четко выявляться «узкие места», а их улучшения должны стать основными целями следующего этапа (или этапов). Старайтесь найти возможность для принципиального тестирования методологии, прежде чем браться за сколько-нибудь значительный объем работы – даже для отдельных компонентов каждого этапа.
3. Полностью запрограммируйте каждый этап и проведите системное тестирование всего прототипа; не поддавайтесь искушению слишком быстро перейти к следующему этапу. Обязательно проведите полные системные тесты на каждом этапе – полученные результаты будут использованы для проектирования следующего этапа.
4. После завершения очередной итерации вернитесь к пункту 2. Как бы далеко ни зашла работа над проектом, не бойтесь переписывать целые компоненты с самого начала.

В академической среде, где коммерческое давление практически полностью отсутствует, и хирургические методы пункта 4 применяются гораздо проще. В коммерческих средах подобная «необходимая жестокость» требует изрядной смелости, однако мы считаем, что нерешительность в критический момент является одной из важных, но часто недооцениваемых причин неудач проектов – особенно самых новаторских и сложных.

Если вы хотите построить красивую архитектуру, зачастую обладающую незапланированными достоинствами, придерживайтесь своих убеждений и ничего не бойтесь.

	Принципы и свойства		Структуры
	Гибкость	✓	Модуль
	Концептуальная целостность		Зависимость
✓	Возможность независимого изменения	✓	Обработка
	Автоматическое распространение		Доступ к данным
✓	Удобство построения		
	Адаптация к росту		
	Сопротивление энтропии		

10

Метациклические виртуальные машины: Jikes RVM

Иэн Роджерс
Дэйв Гроув

Похоже, выполнение кода в управляемой среде времени выполнения становится одним из господствующих течений в среде современных разработчиков. Большая часть всего создаваемого кода предназначена именно для управляемых сред времени выполнения. Но, несмотря на возрастающую популярность сред времени выполнения, они чаще всего пишутся не на том языке, который поддерживают сами. Виртуальные машины Java, выполняющие функции среды времени выполнения для Java-приложений, обычно реализуются на языках программирования C и C++.

В этой главе мы представим обзор виртуальной машины Jikes RVM, написанной на Java для запуска Java-приложений. На Java написана не только система времени выполнения, но и другие компоненты архитектуры. К их числу относится адаптивная и оптимизирующая системы компиляции, поддержка многопоточности, обработка исключений и уборка мусора. Мы представим сводку таких систем и объясним, почему единое представление о языке, среде времени выполнения и реа-

лизации помогает строить более удобные (а возможно, и более оптимальные) системы.

Предыстория

Методология разработки новых языков программирования является одним из основных вопросов теоретического программирования. В этой области существуют свои формальные модели – например, T-диаграммы (Aho, 1986). На T-диаграмме, показанной на рис. 10.1, изображен компилятор C, который состоит из машинного кода PowerPC и создает машинный код PowerPC. Этот компилятор создает компилятор Pascal, написанный на C, который генерирует машинный код PowerPC. В итоге мы получаем компилятор, который состоит из машинного кода PowerPC и создает машинный код PowerPC.

В отличие от традиционных языков программирования, компилируемых в машинный код компьютера, на котором должна запускаться программа, многие современные языки компилируются в архитектурно-нейтральный машинный код (в Java он называется байт-кодом). Нейтральный машинный код позволяет портировать приложение на любую платформу, в которой присутствует соответствующая среда времени выполнения. Таким образом, код Java может выполняться на любой платформе, на которой имеется виртуальная машина Java.

Современные языки стремятся помочь программисту, устраняя саму потенциальную возможность ошибок программирования на архитектурном уровне. Самым очевидным примером является безопасная

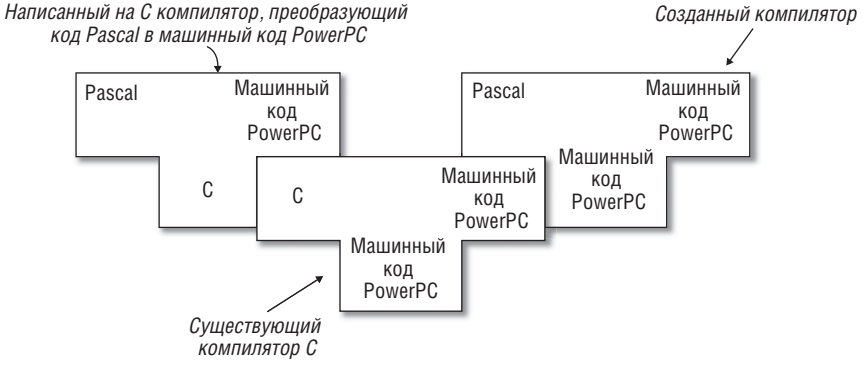


Рис. 10.1. *T-диаграмма создания компилятора, преобразующего код Pascal в машинный код PowerPC*

работа с памятью; возможности программиста по работе с типами данных ограничиваются, а освобождение памяти может осуществляться только в ходе автоматизированной уборки мусора. Другой пример – возможность инициирования исключений.

Одним из важных принципов проектирования языков программирования является *самодостаточность*. Иначе говоря, язык программирования должен обладать достаточными выразительными возможностями, чтобы компилятор с этого языка можно было написать на самом языке. Например, компилятор Pascal, написанный на Pascal, самодостаточен, тогда как компилятор Pascal, написанный на C, этим свойством не обладает. Самодостаточность позволяет разработчику компилятора использовать функции языка, для которого компилятор предназначен. Самодостаточность формирует полезную обратную связь: создатели реализации языка желают использовать нетривиальные и/или выразительные языковые возможности в частях реализации, критичных по быстродействию, и при этом часто обнаруживают новые способы эффективной реализации упомянутых языковых возможностей.

Несмотря на очевидную полезность самодостаточности компилятора, многие среды времени выполнения не пишутся на тех языках, на которых они обычно работают. Например, среда, написанная на C или C++, используется для запуска приложений, написанных на Java. Если в среде присутствует ошибка, связанная с безопасностью памяти, то она приведет к сбою Java-приложения, хотя само приложение безопасно работает с памятью. Устранение подобных ошибок – важная причина для создания самодостаточных сред времени выполнения.

С течением времени компьютерные системы развиваются, а мы начинаем лучше понимать их; соответственно могут изменяться и требования к языкам программирования. Например, в языках программирования C и C++ не существует стандартной библиотеки для реализации многопроцессорной многопоточности (хотя существуют популярные расширения, такие как POSIX threads и OpenMP). У современных языков эти функции встраиваются в основной синтаксис языка или в стандартную библиотеку. Возможность использования усовершенствованных библиотек и абстракций в среде времени выполнения – другая важная причина для использования самодостаточных сред.

Наконец, в системе должна существовать коммуникационная прослойка, через которую среда выполнения взаимодействует с выполняемым приложением. В частности, этой коммуникационной прослойке приходится передавать объекты, приводя формат одного языка программирования к формату другого языка. Коммуникационная прослойка также должна помнить о том, что объекты, используемые за пределами

управляемой среды выполнения, не должны уничтожаться уборщиком мусора. В самодостаточных средах такие коммуникационные прослойки не нужны (по крайней мере, во многих ситуациях).

Надеюсь, мы привели достаточно веские причины для использования самодостаточных сред времени выполнения. В этой главе будет представлен обзор одной из таких сред – Jikes RVM; среда написана на Java и используется для выполнения Java-приложений. Самодостаточные среды времени выполнения называются *метациклическими* средами (Abelson et al., 1985). Метацикличность Jikes RVM не уникальна; ее создатели черпали вдохновение в сходных системах, таких как Lisp (McCarthy et al., 1962) и виртуальная машина Squeak for Smalltalk (Ingalls et al., 1997). Так как метациклическая виртуальная машина написана на Java, в нашем распоряжении были многие замечательные инструменты, среды разработки и библиотеки. Но в некоторых сообществах к Java относятся с предубеждением, поэтому сначала мы должны развеять некоторые мифы, из-за которых люди полагают, что в самой природе метациклической среды Java уже заложены изначальные недостатки.

Мифы, связанные со средами времени выполнения

Вопрос о том, как лучше всего создавать приложения для разных платформ, до сих пор активно обсуждается. Приходится учитывать такие факторы, как ресурсы, доступные на месте выполнения приложения, производительность труда разработчиков, занятых созданием приложения, и зрелость среды разработки. Если приложение и среда разработки строятся по общим правилам, то требования к быстродействию и затратам памяти также относятся и к среде разработки. Мы должны опровергнуть некоторые расхожие мифы по поводу управляемых сред.

Компиляторы времени выполнения должны работать быстро, поэтому они должны быть простыми

Одно из распространенных заблуждений по поводу сред времени выполнения состоит в том, что они имеют дело исключительно с JIT-компиляцией. Действительно, JIT-компиляция должна создавать код быстро, потому что он будет использоваться сразу же по готовности. Хотя эта простая модель выполнения использовалась во многих ранних JVM и прототипах сред времени выполнения, современные реально используемые виртуальные машины используют ту или иную форму из-

бирательной оптимизации. При избирательной оптимизации посредством профилирования выделяется подмножество методов, которые должны компилироваться с применением агрессивной оптимизации; остальные методы либо интерпретируются, либо компилируются очень быстрым неоптимизирующим компилятором непосредственно перед выполнением. Именно избирательность позволяет использовать сложные оптимизирующие компиляторы на стадии выполнения.

Неограниченный анализ в статическом компиляторе непременно обеспечивает лучшее быстродействие

Среды времени выполнения интенсивно используются многими приложениями, поэтому их оптимизация для обеспечения максимально возможного быстродействия по всем приложениям выглядит логично. Тем не менее многие оптимизации могут применяться только в том случае, если среда времени выполнения создавалась как часть динамического окружения.

Профилирование

Многие аспекты среды могут изменяться в ходе выполнения – например, средний размер блоков данных или применение разных программных решений, основанных на разных паттернах проектирования. Оперативное профилирование позволяет своевременно использовать эту информацию для сокращения затрат (например, прогнозирование переходов), а также открывает возможности для более мощных оптимизаций (таких как прогнозирование значений). Пример прогнозирования значений в Java: разумно предположить, что большая часть потоковых операций вывода будет выполняться с файловым потоком `java.lang.System.out`. Прогнозирование значений является расширением приема неполного вычисления, который будет описан ниже (см. раздел «Неполное вычисление»).

Разные конфигурации системы

Спектр систем, в которых может выполняться приложение, постоянно расширяется. Особые возможности разных процессоров, объем памяти, количество процессоров, требования к потребляемой мощности, текущая загрузка системы – все эти факторы важны для понимания того, как среда времени выполнения должна адаптироваться к текущей задаче.

Межпроцедурный анализ

Межпроцедурный анализ – важная функция оптимизирующего компилятора, которая выводит оптимизацию за границы отдельных методов. Статический анализ не ограничен по времени, но он часто

порождает столько данных, что компилятор не может определить, какие данные действительно важны. Данные обратной связи, полученные в ходе выполнения, более оперативны, поэтому они дают более надежную информацию для управления межпроцедурным анализом и другими оптимизациями компилятора.

Анализ времени выполнения расходует слишком много ресурсов

Действительно, для работы среды времени выполнения требуется дополнительная память (по аналогии с использованием стандартных библиотек в традиционных приложениях). Вдобавок среда времени выполнения должна хранить служебную информацию, которая используется при компиляции и выполнении программ. Впрочем, эти затраты памяти весьма умеренны, а своевременный и эффективный сбор данных позволяет извлечь максимум пользы с минимальными затратами.

Динамическая загрузка классов снижает быстродействие

Многие современные среды времени выполнения, включая Java, поддерживают возможность динамического расширения. Данная возможность может пригодиться для объединения частей системы, полученных из разных источников; например, в случае Java компоненты могут загружаться из Интернета. Польза динамического расширения несомненна, однако оно означает, что компилятор не может делать некоторые предположения, которые были бы возможны при наличии всей необходимой информации. Например, если при вызове метода объект может относиться только к одному возможному классу, лучше избежать динамической диспетчеризации метода и вызвать метод напрямую. Для подобных ситуаций в оптимизирующих средах существуют специальные приемы оптимизации, которые будут описаны позднее, в разделе «Замена в стеке».

Уборка мусора работает медленнее ручного управления памятью

Автоматическая уборка мусора относится к числу передовых технологий программирования. Память запрашивается приложением, а затем освобождается, когда необходимость в ней отпадет. В схеме с ручным управлением памятью приложение запрашивает и освобождает память, явно выдавая соответствующие команды в среде времени выпол-

нения. Хотя ручное управление памятью часто становится причиной ошибок, бытует мнение, что оно необходимо для нормального быстродействия. Однако сторонники такого мнения упускают из виду многие затруднения, возникающие из-за ручного управления памятью. В частности, программисту приходится следить за тем, какие блоки памяти используются в данный момент, и составлять списки неиспользуемых блоков. Из-за возможности параллельных запросов памяти из разных программных потоков приходится решать проблемы фрагментации памяти и объединять мелкие блоки, чтобы выделить более крупный блок, – программирование менеджера памяти становится весьма нетривиальной задачей. Ручное управление памятью также не позволяет перемещать объекты в памяти (например, для устранения фрагментации).

Требования менеджера памяти зависят от специфики конкретного приложения. В контексте метациклической среды простому JIT-компилятору обычно не нужно выполнять сложные операции с памятью, поэтому для него достаточно хорошо подойдет как ручное управление памятью, так и автоматическая уборка мусора. Для более сложного оптимизирующего компилятора ситуация не столь очевидна – не считая того факта, что уборка мусора очевидно сокращает риск ошибок. В других частях среды времени выполнения возникают свои трудности, которые будут описаны позднее в разделе «Волшебство и аннотации». Уборка мусора действительно может работать медленнее ручного управления памятью – с этим трудно спорить, но она определенно улучшает гибкость настройки системы.

Резюме

Так как инструменты разработчика сами по себе являются приложениями, наш исходный вопрос – как лучше разрабатывать приложения – приобретает рекурсивную природу. Управляемые языки устраняют наши ошибки на архитектурном уровне и повышают производительность труда разработчика. Упрощение модели разработки, выявление новых возможностей для оптимизации приложения и среды – метациклический подход к этим проблемам позволяет разработчику извлечь максимум пользы из существующих возможностей без создания дополнительных барьеров между разными представлениями системы (приложения, среды времени выполнения и компилятора). В следующих разделах будет описана Jikes RVM – среда времени выполнения, которая объединяет эти принципы.

Краткая история Jikes RVM

Jikes RVM происходит от проекта IBM, который назывался Jalapeño. Проект Jalapeño был запущен в ноябре 1997 года с целью разработки гибкой исследовательской инфраструктуры для анализа новых идей из области проектирования высокопроизводительных виртуальных машин. В начале 1998 года появился исходный рабочий прототип, на котором могли запускаться небольшие Java-программы. Весной 1998 года началась работа над оптимизирующим компилятором, и размер проекта стал резко расти. К началу 2000 года участники проекта опубликовали несколько научных статей с описанием различных аспектов Jalapeño, а университетские исследователи стали выражать интерес в получении доступа к системе, чтобы использовать ее в качестве основы для собственных исследований.

К моменту перехода на модель распространения с открытым кодом в октябре 2001 года Jikes RVM использовалась по лицензии от IBM уже в 16 университетах. Сообщество быстро расширялось; в него входили сотни исследователей из более чем 100 учреждений. Машина Jikes RVM стала темой 188 статей, появившихся в публикациях с коллегиальным рассмотрением, а также заложила основу для более 36 университетских диссертаций.

Версия 2 представляла собой исходный вариант Jikes RVM с открытым кодом; она поддерживала архитектуры как Intel, так и PowerPC. Были доступны реализации разных алгоритмов уборки мусора, включая алгоритмы с подсчетом ссылок, пометки с удалением и полупространства. Годом позже была выпущена версия Jikes RVM 2.2. Одним из ее главных усовершенствований стала абсолютно новая реализация подсистемы управления памятью, названная Memory Management Toolkit (MMTk).

Инфраструктура MMTk получила очень широкое распространение в сообществе уборки мусора и была портирована в другие среды, помимо Jikes RVM. MMTk и методы уборки мусора рассматриваются ниже в разделе «Уборка мусора». Оптимизирующий компилятор и система адаптивной оптимизации тоже были существенно доработаны, а разработка среды времени выполнения упростилась благодаря переходу на GNU Classpath – стандартные библиотеки классов с открытым кодом. Версия Java RVM 2.2.1 (апрель 2003 г.) стала одной из первых сред времени выполнения Java с открытым кодом, в которых работала большая часть Eclipse IDE.

Между версиями 2.2 и 2.4.6 прошло почти четыре года. За это время было сделано много значительных усовершенствований в функцио-

нальности и быстродействии, но структура и архитектура исходного кода почти не изменились.

Версия Jikes RVM 3.0, выпущенная в августе 2008 года, стала результатом почти двухлетних совместных усилий сообщества по модернизации и усовершенствованию системы. В кодовую базу были включены новые возможности языка Java 5.0, система сборки перешла на Apache Ant, а значительно усовершенствованная инфраструктура тестирования повышала стабильность и быстродействие системы. В результате многочисленных усовершенствований функциональности и оптимизаций многие программы достигли быстродействия, сопоставимого с быстродействием современных продуктивных JVM (реализованных в традиционных языках с системой времени выполнения, таких как C/C++).

Над Jikes RVM работало слишком много людей, чтобы их можно было упомянуть всех по отдельности, поэтому мы выражаем благодарность всему сообществу разработки Jikes RVM за проделанную работу. Менее 100 человек предоставило свой код для включения в Jikes RVM, а 19 человек образовали основную группу Jikes RVM. Полный список благодарностей опубликован на сайте Jikes RVM. О начальных этапах существования Jikes RVM и росте сообщества при переходе на модель с открытым кодом более подробно рассказано в статье 2005 IBM System Journal (Alpern et al., 2005).

Инициализация самодостаточной среды времени выполнения

По сравнению с инициализацией модели с традиционным компилятором (рис. 10.1) инициализация метациклической среды происходит чуть сложнее. T-диаграмма процесса изображена на рис. 10.2.

Загрузочный образ состоит из нескольких файлов, представляющих состояние памяти в самодостаточной системе (правая «буква T» на рис. 10.2). Загрузочный образ содержит код и данные, аналогичные тем, которые содержатся в объектном файле обычного компилятора. Дополнительная секция загрузочного образа Jikes RVM содержит корневую карту, создаваемую уборщиком мусора. Корневые карты описываются ниже, в разделе «Уборка мусора». *Построитель загрузочного образа* использует компиляторы Jikes RVM для создания файлов загрузочных образов, выполняемых на инициализирующей виртуальной машине Java. Загрузчик отвечает за размещение загрузочного образа в правильной области памяти; в Jikes RVM загрузчик называется *исполнителем загрузочного образа*.

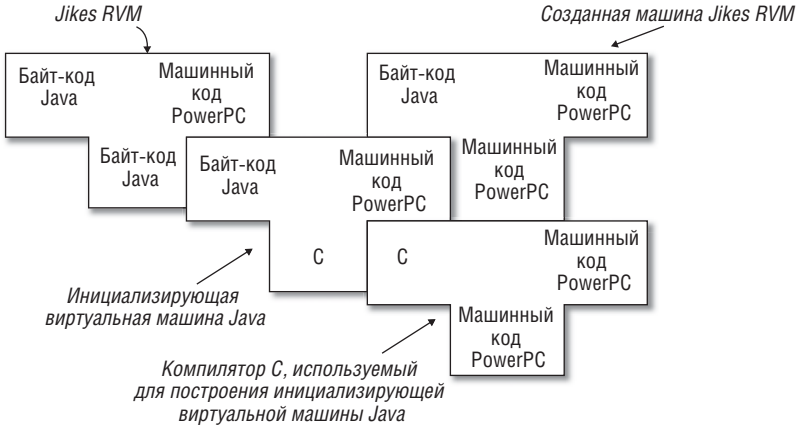


Рис. 10.2. *Процесс инициализации Jikes RVM в существующей виртуальной машине Java, написанной на C*

Структура объектов

Построитель загрузочного образа должен расположить объекты на диске так, как они будут использоваться в работающем экземпляре Jikes RVM. Конфигурация объектной модели в Jikes RVM может изменяться, позволяя анализировать разные архитектурные альтернативы с сохранением фиксированной структуры остальных компонентов системы. Примером архитектурной альтернативы может служить выделение дополнительных битов на уровне объекта для хеширования объектов или выделение дополнительных битов для реализации быстрых блокировок для синхронизации. Общая структура объектов Jikes RVM показана на рис. 10.3.

В стандартной 32-разрядной объектной модели Jikes RVM заголовок объекта сейчас состоит из двух слов: первое содержит ссылку на блок *TIB* (Type Information Block), а во втором хранится информация состояния для блокировки, хеширования и уборки мусора. За заголовком объекта следуют поля объекта. Для массива в первом поле указывается его длина, а в остальных – элементы массива. Чтобы избежать применения смещения при обращениях к массиву, размер которого равен размеру заголовка объекта вместе с полем длины массива, все ссылки на объекты смещены на три слова от начала объекта. При такой схеме адресации нулевой элемент массива начинается с нулевым смещением в объекте, но с другой стороны, заголовок объекта всегда

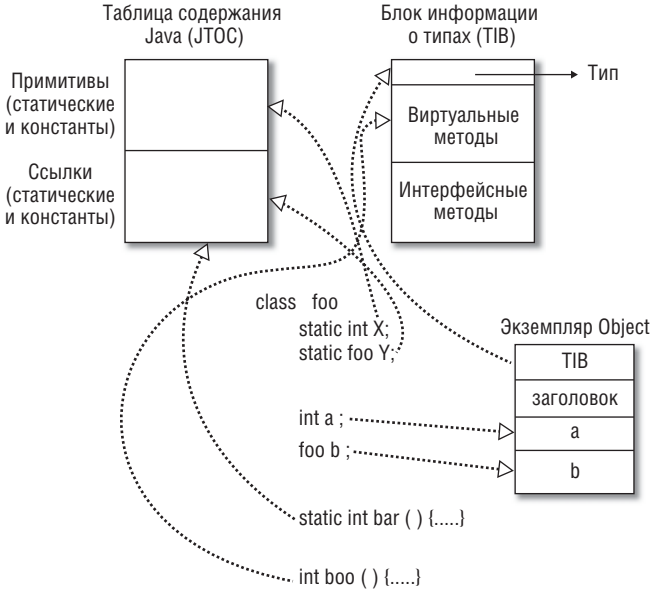


Рис. 10.3. *Структура объектов Jikes RVM*

отстает на три слова от ссылки, а первое поле объекта всегда располагается с отрицательным смещением от ссылки на объект.

Блок TIB предназначен для хранения данных, общих для всех объектов конкретного типа. Эти данные используются в основном для диспетчеризации виртуальных методов и методов интерфейсов (диспетчеризацией называется процесс определения метода, который должен быть вызван для конкретного объекта). Методы классов размещаются в определенных позициях TIB для быстрой и эффективной диспетчеризации. В TIB также хранится информация для быстрого получения данных о типе, ускоряющей операции Java `instanceof` и `checkcast`, а также специальные методы для работы с объектом в ходе уборки мусора.

Помимо объектов среда также должна отслеживать статические данные, принадлежащие классам. Статические данные хранятся в специальной таблице *JTOC* (Java Table Of Contents). Таблица JTOC расширяется в двух направлениях: с положительными смещениями в JTOC хранится содержимое статических полей, содержащих ссылки и ссылочные литералы. Литеральные значения в Java отчасти напоминают строковые литералы: это данные, напрямую доступные для байт-кода, но не хранящиеся в полях объектов. Отрицательные адреса в JTOC

предназначены для хранения значений примитивных типов. Подобное разделение позволяет уборщику мусора легко определить, какие ссылки из статических полей не позволяют считать объект кандидатом для уничтожения.

Структура памяти среды времени выполнения

Построитель загрузочного образа отвечает за формирование структуры памяти виртуальной машины при ее первой загрузке. Все объекты, необходимые для загрузочного образа, находятся на своих местах, потому что ссылки на них присутствуют в коде запуска приложения Java. На рис. 10.4 показаны основные области памяти, используемые при выполнении Jikes RVM.

Исполнитель загрузочного образа

Исполнитель загрузочного образа и его стек составляют загрузчик, ответственный за размещение образа в памяти.

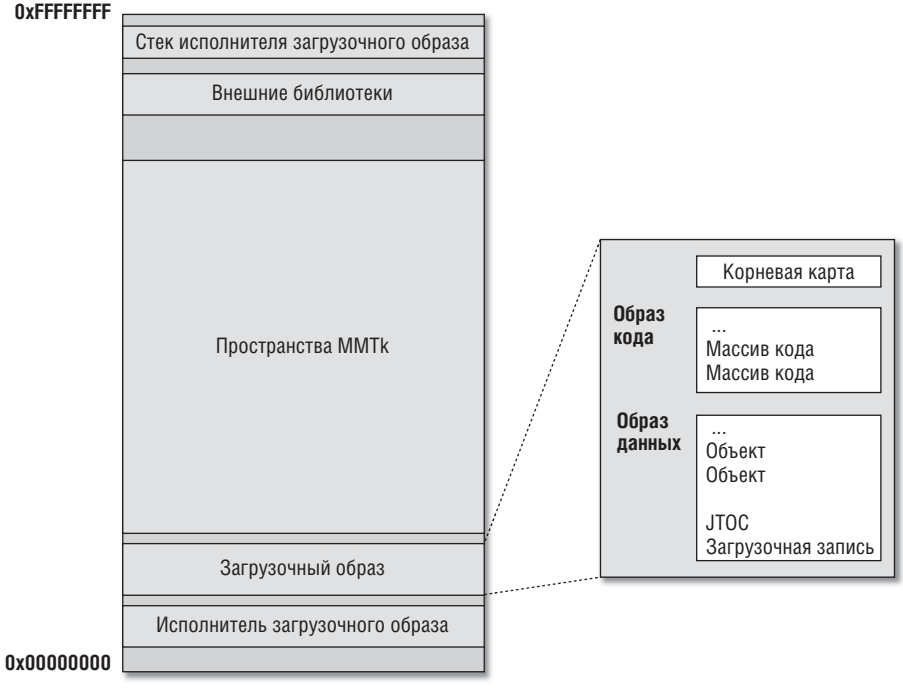


Рис. 10.4. *Структура памяти Jikes RVM во время выполнения*

Внешние библиотеки

Память также необходимо выделить и для внешних библиотек, используемых библиотекой классов. За дополнительной информацией обращайтесь к разделу «Внешний интерфейс».

Пространства MMTk

В этой области находятся различные кучи, используемые MMTk для обеспечения работы приложения. Содержимое освобождается в ходе уборки мусора.

Корневая карта

Предназначенная для уборщика мусора информация о полях, доступных из загрузочного образа. За дополнительной информацией обращайтесь к разделу «Уборка мусора».

Образ кода

Исполняемый код статических и виртуальных методов, доступных непосредственно из JTOC или блоков TIB соответственно. Код записывается в отдельную область загрузочного образа для обеспечения защиты памяти.

Образ данных

При формировании образа данных сначала записывается загрузочная запись и JTOC, а затем выполняется перебор объектов, доступных из JTOC. Для перебора объектов в загрузочной JVM используется механизм рефлексии Java. Порядок перебора объектов может повлиять на их относительное расположение, а следовательно, и на быстродействие, поэтому механизм перебора может настраиваться (Rogers et al., 2008).

JTOC

Как упоминалось ранее в разделе «Структура объектов», в JTOC хранятся литералы и значения статических полей. Загрузочный образ строится перебором объектов от JTOC.

Загрузочная запись

Таблица в начале образа данных, которая содержит общие данные для исполнителя загрузочного образа и Jikes RVM. Эти значения обычно не могут определяться динамически в ходе загрузки.

Компиляция первичных классов и заполнение JTOC

Первичными классами (*primordials*) называются классы, которые должны быть встроены в загрузочный образ для работы последнего. Самым важным первичным классом является класс `org.jikesrvm.VM`,

ответственный за запуск виртуальной машины. Если класс не является частью загрузочного образа (то есть не относится к категории первичных), то при обращении к нему во время выполнения загрузчик классов производит загрузку и компоновку класса.

Список первичных классов строится в ходе инициализации посредством поиска по каталогам и чтением списка компилируемых классов. Список особенно важен для типов-массивов. Теоретически список первичных классов можно было бы построить посредством многократной компиляции и расширения набора классов, включаемых в загрузочный образ, но это привело бы к значительному увеличению времени построения Jikes RVM. Чтобы решить эту проблему, в Java предлагается использовать аннотации Java для пометки первичных классов.

Прежде чем переходить к перебору графа объектов и записи загрузочного образа, построитель загрузочного образа компилирует первичные классы. Чтобы откомпилировать первичный класс, необходимо загрузить его загрузчиком классов Jikes RVM, что приведет к автоматическому выделению необходимой памяти в JTOC и TIB, а затем перебрать все методы и откомпилировать их одним из компиляторов Jikes RVM. Так как при этом мы имеем дело исключительно с обычным кодом Java, построитель загрузочного образа по возможности использует многопоточный API язык Java для параллельного выполнения этой задачи.

После того как основной набор первичных классов будет откомпилирован, граф объектов в куче управляющей JVM представляет функциональность, достаточную для того, чтобы экземпляр Jikes RVM мог инициализироваться, создать дополнительные объекты и начать загрузку и выполнение пользовательских классов. В завершение процесса инициализации происходит перебор графа основных объектов и запись его на диск в объектной модели Jikes RVM с использованием возможностей механизма рефлексии Java, поддержка которого обеспечивается хостовой виртуальной машиной.

Исполнитель загрузочного образа и VM.boot

Как упоминалось ранее в разделе «Инициализация самодостаточной среды времени выполнения», исполнитель загрузочного образа отвечает за загрузку откомпилированных образов в память. Конкретные детали зависят от операционной системы, но образы настраиваются для страничной загрузки в память по требованию (иначе говоря, страницы из загрузочного образа остаются на диске до тех пор, пока не потребуются).

Оказавшись в памяти, исполнитель загрузочного образа инициализирует загрузочную запись и заполняет регистры машины для передачи управления методу Jikes RVM `org.jikesrvm.VM.boot` (далее сокращенно `VM.boot`). Jikes RVM отвечает за формирование структуры памяти, за эффективную уборку мусора и организацию стека, эффективную при обработке исключений Java (см. «Модель исключений» ниже). После входа в метод `VM.boot` необходимы специальные «обертки» для передачи управления между внешним кодом исполнителя загрузочного образа и библиотеками C (см. далее раздел «Внешний интерфейс»).

Задача `VM.boot` – обеспечить готовность виртуальной машины к выполнению программы. Для этого метод инициализирует компоненты RVM, которые не могли быть инициализированы на момент записи загрузочного образа. Некоторые компоненты (например, уборщика мусора) приходится запускать явно. Остальные компоненты образуют небольшое подмножество первичных классов, которые не были полностью записаны в загрузочный образ. Для инициализации таких классов должен быть выполнен статический инициализатор класса.

Инициализация потоковой системы является важной частью метода `VM.boot`. Она создает необходимые программные потоки уборки мусора, поток для выполнения методов-завершителей (finalizers) объектов, а также потоки, ответственные за систему адаптивной оптимизации. Отладочный поток тоже создается, но планируется к выполнению только в случае получения Jikes RVM сигнала от операционной системы. Последним создается и запускается главный поток, отвечающий за выполнение приложения Java.

Компоненты времени выполнения

В предыдущем разделе был описан процесс перевода Jikes RVM в состояние готовности к работе. В этом разделе мы рассмотрим основные компоненты времени выполнения Jikes RVM, начиная с тех, которые непосредственно отвечают за выполнение байт-кода Java, а затем рассмотрим другие подсистемы виртуальных машин, обеспечивающие выполнение.

Базовая модель выполнения

Jikes RVM не имеет интерпретатора; все байт-коды должны быть предварительно преобразованы одним из компиляторов Jikes RVM в «родной» машинный код. Единицей компиляции является метод; компиляция методов откладывается до первой активизации программой. Исходная компиляция осуществляется базовым компилятором Jikes

RVM – простым неоптимизирующим компилятором, способным очень быстро генерировать неэффективный код. В ходе выполнения адаптивные системные мониторы Jikes RVM программируются на выявление активных участков программ, которые избирательно перекомпилируются оптимизирующим компилятором Jikes RVM. Этот существенно более совершенный компилятор генерирует более качественный код, но за счет существенно больших затрат времени компиляции и памяти по сравнению с базовым компилятором.

Модель избирательной оптимизации не уникальна для Jikes RVM. Все современные продуктивные JVM используют ту или иную разновидность избирательной оптимизации для того, чтобы направить ресурсы оптимизирующей компиляции на подмножество методов программы, в котором они приносят наибольшую пользу. Как упоминалось ранее, избирательная оптимизация является ключом к работе сложных оптимизирующих компиляторов в качестве динамических компиляторов.

Система адаптивной оптимизации

С архитектурной точки зрения система адаптивной оптимизации реализуется в виде набора нежестко синхронизированных сущностей. Так как система реализована на Java, для структурирования кода могут использоваться встроенные средства языка, такие как программные потоки и мониторы.

В ходе выполнения программы данные, собранные по таймеру, накапливаются в буферах выполняемых программных потоков Java. Собираются два вида профильных данных: для текущего выполняемого метода (с целью выявления кандидатов на применение оптимизирующей компиляции) и для стека вызовов (с целью выявления важных ребер графа вызовов для профильной подстановки кода). Когда буфер заполняется, низкоуровневый профильный агент отправляет сигнал *организатору* более высокого уровня (реализованному в виде отдельных потоков Java) для обобщения и сохранения необработанных профильных данных.

Периодически поток-*контроллер* анализирует текущие профильные данные и использует аналитическую модель для определения методов, которые должны быть запланированы для оптимизирующей компиляции. Эти решения принимаются с применением стандартного двухконкурентного варианта решения проблемы «проката лыж[1]» из онлайновых алгоритмов. Метод не выбирается для оптимизации, пока

[1] За дополнительной информацией обращайтесь по адресу *http://en.wikipedia.org/wiki/Ski_rental_problem*.

предполагаемый выигрыш (повышение скорости при будущих вызовах) после оптимизации не превысит ожидаемые затраты (время компиляции). Оценка соотношения затрат/выигрыша производится на основании объединения профильных данных (как часто метод-кандидат поставлял данные в текущем выполнении?) с эмпирически выведенными константами, описывающими ожидаемое относительное ускорение и затраты компиляции для каждого уровня оптимизации оптимизирующего компилятора.

Оптимизирующая компиляция

Являясь метациклической средой, Jikes RVM для обеспечения хорошего быстродействия компилирует себя вместо того, чтобы полагаться на другой компилятор. Метацикличность создает позитивную обратную связь: наше сильное желание писать чистый, элегантный и эффективный код Java в реализации виртуальной машины привело нас к разработке новаторских оптимизаций компилятора и приемов реализации времени выполнения. В этом разделе будет описан оптимизирующий компилятор, который состоит из многих фаз, разделенных на три основные стадии:

1. Высокоуровневое промежуточное представление (HIR, High-level Intermediate Representation).
2. Низкоуровневое промежуточное представление (LIR, Low-level Intermediate Representation).
3. Промежуточное представление машинного уровня (MIR, Machine-level Intermediate Representation).

Все стадии работают с графом управляющей логики, состоящем из базовых блоков, которые в свою очередь состоят из списков команд (рис. 10.5). Команда состоит из операндов и оператора. Так как оператор постепенно все сильнее привязывается к конкретной машине, оператор может изменяться (мутировать) во время жизненного цикла команды. Операнды делятся на определяемые и используемые. Основные разновидности операндов составляют константы и операнды с кодированием регистров. Базовый блок представляет собой список команд, в котором команды перехода могут находиться только в конце списка. Специальные команды отмечают начало и конец базового блока. В графе управляющей логики разные области кода соединяются ребрами. Исключения интерпретируются особым образом (см. далее «Факторизация графа управляющей логики»). Так как все три главные стадии компиляции используют одно базовое промежуточное представление, многие оптимизации применяются более чем в одной стадии. Основные задачи трех стадий компиляции описываются в следующем разделе.

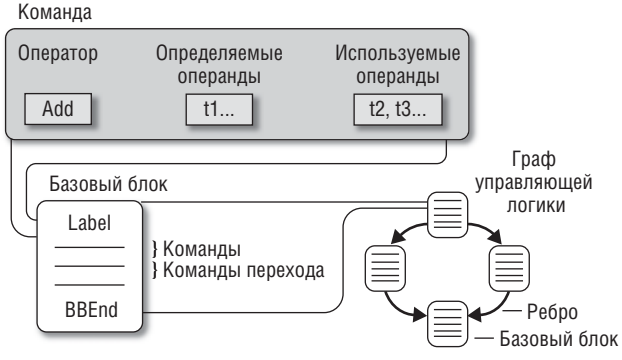


***Рис. 10.5**. Промежуточное представление оптимизирующего компилятора*

HIR

Высокоуровневое промежуточное представление (HIR) генерируется в фазе компиляции, называемой BC2IR, на вход которой поступают байт-коды. В этой начальной фазе выполняется обработка на основании стека операндов байт-кода. Вместо генерирования отдельных операций стек имитируется, а байт-коды объединяются для создания команд HIR. Операторы уровня HIR эквивалентны операциям, выполняемым в байт-коде, но используют неограниченное количество символических регистров вместо стека выражений.

Сформированные команды сокращаются до более простых операций (если последние существуют, разумеется). Далее анализируется возможность подстановки оставшихся команд вызова. Основная часть фазы BC2IR написана рекурсивно, поэтому при подстановке выполняется рекурсивное использование фазы BC2IR.

В фазе HIR выполняются *локальные оптимизации*, то есть оптимизации, направленные на сокращение сложности базового блока. Так как анализ ограничивается базовым блоком, рассматриваются только зависимости внутри этого блока. Это избавляет фазу компиляции от необходимости учитывать эффект переходов и циклов, которые могут формировать различные зависимости данных. Некоторые виды локальных оптимизаций:

Распространение констант

Распространение значений констант предотвращает их размещение в регистрах.

Распространение копирования

Замена копий регистров использованием исходного регистра.

Упрощение

Переход на менее сложные операции на основании операндов.

Свертка выражений

Свертка деревьев операций. Например, в последовательности «x=y+1; z=x+1» выражение «y+1» сворачивается в «z=x+1», в результате мы получаем «x=y+1; z = y+2».

Исключение общих подвыражений

Поиск команды, выполняющей ту же операцию с теми же операндами, и удаление последней с подстановкой копии результата первой.

Исключение неиспользуемого кода

Если значение регистра определяется, а затем переопределяется без промежуточного использования, то команда исходного определения является неиспользуемым («мертвым») кодом и может быть удалена из программы.

Переходные оптимизации направлены на улучшение графа управляющей логики, чтобы наиболее вероятные пути были расположены оптимальным для компилятора образом. Другие переходные оптимизации удаляют лишние проверки и выполняют раскрутку циклов.

Анализ обращений проверяет, возможно ли обращение к объектам исключительно из контекста компилируемого кода, а также возможно ли совместное использование объекта, обращения к которому не ограничиваются локальным контекстом, между разными программными потоками. Если все обращения к объекту выполняются исключительно локально, то память для объекта можно не выделять, а поля объекта переместить в регистры. Если обращения к объекту выполняются только в одном программном потоке, можно убрать операции синхронизации.

Некоторые оптимизации, использующие дополнительную информацию из промежуточной формы, описываются далее в разделе «Скалярные и расширенные векторные формы SSA».

LIR

Низкоуровневое промежуточное представление (LIR) преобразует сгенерированные на предыдущем этапе «байт-кодо-подобные» операции в операции, больше напоминающие машинный код. Операции с полями заменяются операциями загрузки и сохранения; такие операторы, как `new` и `checkcast`, расширяются в вызовы функций времени выполне-

ния, которые обеспечивают выполнение этих операций (и к которым может применяться подстановка). Большинство оптимизаций HIR также может применяться и на стадии LIR, хотя это не относится к таким оптимизациям, как анализ обращений.

Вследствие особенностей поддерживаемых операций между LIR для разных архитектур существуют небольшие различия. Как и в случае с командами HIR, в командах LIR может использоваться неограниченное количество символических регистров.

MIR

Создание итогового промежуточного представления машинного уровня (MIR) состоит из трех взаимозависимых, но конкурирующих преобразований подсистемы компиляторов. Преобразования названы конкурирующими, потому что их порядок может определить быстродействие генерируемого машинного кода. В Jikes RVM первым преобразованием является выбор команд – процесс, в ходе которого RISC-подобные команды LIR преобразуются в команды, существующие на реальном компьютере. Иногда для выполнения команды LIR требуется более одной реальной команды; например, операция длинного сложения LIR требует двух команд в 32-разрядных архитектурах. В других случаях анализирующий селектор команд с поиском по шаблону *BURS* (Bottom Up Rewrite System) объединяет несколько команд в одну. На рис. 10.6 изображен пример поиска по шаблону. Для архитектуры Intel IA32 найдены два шаблона: один создает три команды с затратами 41, а другой кодирует операции загрузки и сохранения в операнде, находящемся в памяти, для которой затраты составляют всего 17. Выбирается менее затратный вариант.

После выбора команды неограниченное количество условных регистров необходимо отобразить на конечное количество регистров реальной машины. Этот процесс называется распределением регистров. При нехватке регистров данные могут храниться в стеке. Распределитель регистров должен свести количество замен к минимуму, а также принять во внимание все архитектурные требования – например, что все операции умножения и деления должны выполняться с определенными регистрами.

Распределитель регистров Jikes RVM работает по принципу линейного сканирования, что обеспечивает высокую скорость работы, но с возможным генерированием лишних операций копирования и обращений к памяти. Эта проблема в основном актуальна при небольшом количестве машинных регистров. С увеличением количества регистров и усовершенствованием алгоритма линейного сканирования для неко-

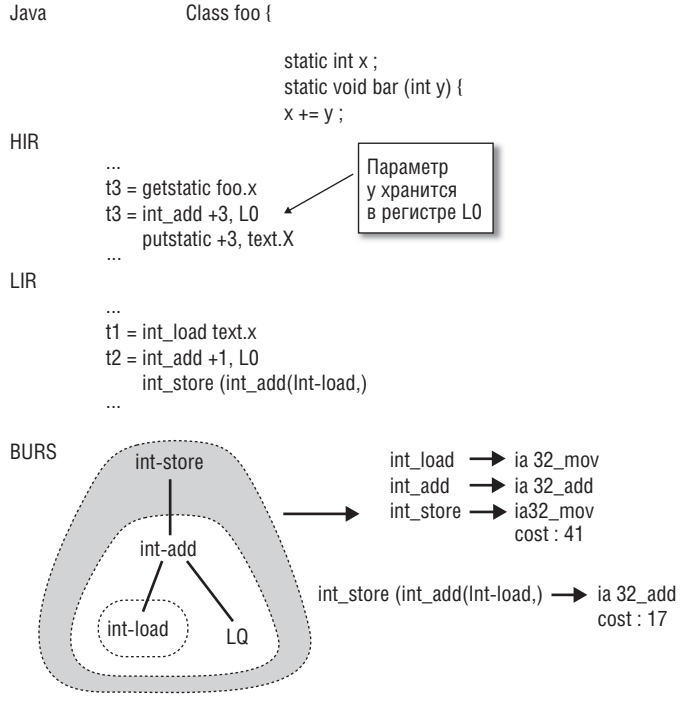


Рис. 10.6. *Выбор команд BURS*

торых видов кода вопрос о том, принесет ли реальный выигрыш более затратная схема распределения регистров, не столь очевиден.

Последней фазой генерирования команд MIR является планирование команд. В этой фазе команды разделяются для того, чтобы процессор мог использовать параллелизм уровня команд. На уровне MIR количество выполняемых оптимизацией невелико. Из-за многочисленных побочных эффектов, связанных с выполнением машинных команд, трудно спрогнозировать, а следовательно, и оптимизировать поведение команды. Другие фазы компиляции уровня MIR связаны с соблюдением правил вызова, обработки исключений и других условных конвенций.

Разложенный граф управляющей логики

Если программа Java выполняет обращение к памяти через `null`-указатель, а также при выходе индекса за границы массива, происходит исключение времени выполнения. Исключения изменяют последова-

тельно выполнения команд, а следовательно, должны завершать базовые блоки. Это приводит к сокращению размеров базовых блоков и, как следствие, к сокращению потенциала локальных оптимизаций (см. выше раздел «HIR»). Для увеличения размера базовых блоков зависимость управляющей логики от исключений времени выполнения преобразуется в искусственные зависимости данных на уровнях HIR и LIR.

Эта зависимость гарантирует правильное упорядочение операций для семантики исключений; таким образом, появляется возможность выхода управления за пределы базового блока в середине его выполнения для обработки исключения. Если промежуточная форма содержит дополнительные точки выхода по исключению в середине блоков, граф управляющей логики называется *факторизованным* (Choi et al., 1999).

Компилятор создает команды, которые явно проверяют исключения времени выполнения и генерируют искусственные «сторожевые» результаты. Команды, требующие определенного упорядочения, используют эту защиту. Команды, которые могут выдать исключение, обозначаются термином PEI (Potentially Exceptioning Instruction). Все PEI-команды генерируют сторожевые результаты, как и команды, которые могут использоваться для удаления PEI в случае их избыточности. Например, ветвь проверки `null` делает проверку `null`-указателя для того же значения избыточной. Сторож ветви используется вместо сторожа проверки `null`-указателя, обеспечивая невозможность изменения порядка команд до завершения ветви.

На рис. 10.7 изображена отдельная операция присваивания элементу массива, с преобразованием соответствующих исключений времени выполнения в PEI, а также со сторожевыми зависимостями, обеспечивающими правильную последовательность выполнения кода.

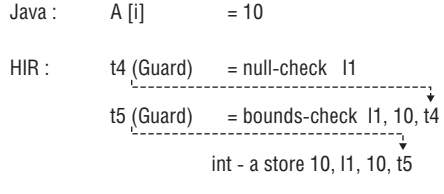


Рис. 10.7. *Пример генерирования команд в факторизованном графе управляющей логики*

Скалярные и расширенные векторные формы SSA

Форма статического одиночного присваивания SSA (Static Single Assignment) сокращает зависимости, которые приходится учитывать компилятору при оптимизации. Эта форма гарантирует, что запись в любые регистры (чаще называемые переменными; в фазах HIR и LIR компиляции Jikes RVM все переменные хранятся в регистрах) производится не более одного раза. Преобразование компилятора должно учитывать три вида потенциальных зависимостей:

Полная зависимость

Данные сначала записываются в регистр, а затем читаются из него.

Зависимость вывода

Данные записываются в регистр, а затем записываются в него снова (при любых преобразованиях вторая запись должна производиться после первой).

Антизависимость

Данные сначала читаются из регистра, а затем записываются в него (при любых преобразованиях запись должна производиться после чтения).

Если запись в регистр гарантированно происходит однократно, зависимости вывода и антизависимости исключаются. Это свойство означает, что описанные выше локальные оптимизации могут применяться глобально. Для циклов и других переходов используются специальные `phi`-команды, кодирующие слияние значений с другими позициями графа управляющей логики.

Векторная форма SSA представляет собой расширение формы SSA, в котором операции чтения и записи определяют специальную переменную, называемую *кучей* (heap). Такая модель обращений к памяти позволяет компилятору сделать вывод, что при двух операциях чтения из одного элемента массива в одноименные кучи вторая операция чтения может быть заменена копией первой. С повторными сохранениями происходит то же самое. Кроме того, векторная форма SSA позволяет реорганизовать обращения к несвязанным кучам – например, для вещественных и целочисленных операций. Векторная форма SSA изначально предназначалась для Fortran; в расширенной векторной форме SSA добавлены такие специфические особенности Java, как невозможность создания псевдонимов (aliases) между полями (Fink et al., 2000).

Кроме того, в форме SSA Jikes RVM конструирует `pi`-команды, размещаемые после использования операнда ветвью. `Pi`-команда использует

тот же операнд, что и ветвь, и присваивает ему новое имя, которое должно использоваться вместо операнда в последующих командах. При помощи `pi`-команд компилятор может вычислить, что ветвь выполняет проверку (допустим, проверку `null`), а все последующие проверки `null` с использованием регистрового результата команды `pi`-команды становятся избыточными. Аналогичным образом могут устраняться проверки границ массивов (Bodik et al., 2000).

Оптимизация проверки версий циклов в форме HIR SSA также устраняет возможные исключения. Этот вид оптимизации выводит код проверки исключений из цикла и явно проверяет возможность возникновения исключений перед входом в цикл. Если исключение возможно, то выполняется версия цикла с кодом, генерирующим исключение. Если исключение невозможно, то выполняется версия цикла без исключений.

Частичное вычисление

Оптимизации HIR, в том числе и оптимизации SSA, позволяют сократить сложность участка кода, однако их возможности могут ограничиваться доступностью константных значений. Значение, взятое из массива, часто не может считаться константным, поскольку в Java эти значения всегда могут изменяться (самый распространенный пример – строки). Jikes RVM вводит расширение Java в виде чистой (pure) аннотации, которое позволяет произвести вычисление метода с аргументами-константами средствами рефлексии на стадии компиляции. Применение аннотаций и рефлексии для этой цели в метациклической среде времени выполнения организуется достаточно просто.

Частичное вычисление позволяет полностью устранить некоторые непроизводительные затраты – например, проверки безопасности времени выполнения. Проверка безопасности обычно перебирает содержимое стека и определяет, откуда был вызван защищенный метод, а затем проверяет наличие необходимых разрешений для его вызова. Так как метод, находящийся в стеке в точке вызова, может быть определен компилятором, это позволяет упростить процедуру идентификации. Непосредственная обработка кода проверки безопасности, или (в случае «чистой» проверки) выполнение рефлексивного вызова, позволяет заранее определить результат и устранить проверку, если она всегда будет давать положительный результат.

Замена в стеке

Заменой в стеке, или *OSR* (On-Stack Replacement), называется процесс замены кода в процессе его выполнения в стеке. Например, если в коде, созданном базовым JIT-компилятором, выполняется долгий цикл,

то этот код стоит заменить оптимизированным кодом, сгенерированным оптимизирующим компилятором.

Работа OSR основана на введении новых байт-кодов (для базового JIT-компилятора) или новых команд (для оптимизирующего компилятора), сохраняющих текущее состояние выполнения метода. После сохранения состояния код может быть заменен вновь откомпилированным кодом, и исполнение продолжится с загрузкой сохраненного состояния.

Итоги

В этом разделе были кратко описаны многие нетривиальные компоненты оптимизирующего компилятора Jikes RVM. Написание этих компонентов на Java заложило в эти системы целый ряд изначальных преимуществ. Многопоточность позволяет организовать параллелизм среди компонентов. Подсистемы спроектированы с учетом потоковой безопасности, что позволяет, например, одновременно запустить несколько потоков компиляции. Другое преимущество связано с использованием обобщенных коллекций Java с их понятными, доступными библиотеками, которые упрощают разработку и позволяют компонентам системы сосредоточить внимание на выполнении своих функций, не отвлекаясь на управление структурами данных.

Модель исключений

Действительно ли исключения представляют исключительные ситуации? Хотя многие программисты исходят из такого предположения, виртуальным машинам приходится оптимизировать выполнение для стандартной ситуации, характерной для эталонных тестов: программа читает данные из входного потока, а затем инициирует исключение при обнаружении искомого шаблона. Этот паттерн встречается как в тесте SPECjvm98[1] jack, так и в DaCapo 2006[2] lusearch.

Давайте сначала разберемся, что необходимо для исключения. Каждое исключение должно создать список методов, находящихся в стеке на момент возникновения исключения (трассировка стека), а затем передать управление `catch`-блоку, обрабатывающему исключение. Исключения `null`-указателей эффективно обрабатываются инициированием сбоев защиты памяти – например, чтением или записью в несуществующую страницу. При возникновении сбоя обработчику передается

[1] *http://www.spec.org/jvm98/*

[2] *http://dacapobench.org/*

адрес сбойной команды. По нему определяется трассировка стека и информация обработчика.

Виртуальная машина, написанная на C, может попытаться применить конвенции вызова C к JIT-откомпилированному коду Java. Проблема заключается в том, что конвенции вызова C не сохраняют информацию о том, какой код выполняется в настоящий момент. Для решения этой проблемы виртуальная машина C может попытаться использовать адрес возврата для вычисления выполняемого метода. Для этого необходимо перебрать все методы виртуальной машины и проверить, содержит ли соответствующий откомпилированный код указанный адрес возврата. Поиск повторяется для каждого метода в стеке.

Так как Jikes RVM управляет размещением объектов в памяти, в стек включается дополнительная информация для ускоренной обработки исключений. В частности, Jikes RVM размещает в стеке идентификатор, который сводит получение информации о методе к простому табличному отображению. Для определения байт-кода, в котором произошло исключение, а также местонахождения обработчика, производятся вычисления с использованием адреса возврата или адреса сбойной команды. Таким образом, скорость обработки исключения пропорциональна глубине стека и не зависит от количества методов в виртуальной машине.

Как и другие виртуальные машины, Jikes RVM по возможности устраняет исключения в ходе оптимизации, в лучшем случае сокращая их до отдельных ветвей. Кроме того, код многих методов подставляется в один метод, так что размещение идентификатора в стеке выполняется не часто (только при переходе между методами, к которым подстановка кода не применялась).

Волшебство и аннотации

Сгенерированный компилятором код содержит команды, которые напрямую обращаются к памяти. Необходимость прямого обращения к памяти возникает и в других ситуациях, например для реализации уборки мусора или при обращении к заголовку объекта для установления блокировки. Для обеспечения возможности сильно типизированных обращений к памяти Jikes RVM использует «волшебную» библиотеку *VM Magic*, которая считается практически стандартным интерфейсом и используется в других проектах JVM. VM Magic определяет классы, которые оформляются либо в виде аннотаций компилятора, расширяющих язык Java, либо в виде неупакованных (unboxed) типов, представляющих обращения к базовой системе памяти. Неупакованный тип `Address` используется для прямых обращений к памяти.

Компиляторы Jikes RVM особым образом обрабатывают неупакованные типы VM Magic. Все вызовы методов с неупакованным типом интерпретируются как прямые операции со значением, длина которого соответствует длине слова. Например, метод `plus` интерпретируется как прибавление размера машинного слова к тому, что обычно является указателем на объект. Так как изменяемое значение не является ссылкой на объект, компилятор сохраняет информацию о том, что места хранения неупакованных типов представляют для уборщика мусора такой же интерес, как и блок памяти, в котором хранится поле примитивного типа (например, `int`).

Типизированных обращений к памяти недостаточно для того, чтобы Jikes RVM заработала без малейших проблем. Необходимы специальные методы, которые сообщают компилятору о том, что выполняется какая-то внутренняя операция, или о необходимости частичного обхода семантики сильной типизации Java. Такие специальные методы существуют в *org.jikesrvm.runtime.Magic*. Примером внутренней операции служит метод вычисления квадратного корня, для которого Java не генерирует байт-код, но во многих компьютерных архитектурах имеется специальная команда, или же функции прямого обращения к полям при выполнении рефлексии. Примером обхода типовой семантики Java является нежелательность преобразований типов во время планирования потоков, так как исключения времени выполнения не разрешены в некоторых ключевых точках.

Все эти «волшебные» операции компилируются не так, как обычные методы. На их месте генерируются методы, но если бы эти методы когда-либо были выполнены, это привело бы лишь к сбою с исключением. Вместо этого компилятор по методу определяет, какая «волшебная» операция выполняется, и находит операции, которые должны быть предоставлены для нее компилятором. Во время создания загрузочного образа предоставляются заменители для некоторых неупакованных типов и методов. Например, адреса, находящиеся в загрузочном образе, остаются неизвестными до момента размещения объектов в памяти. С неупакованными типами загрузочных образов связываются идентификаторы и информация о том, каким объектам они соответствуют. В ходе создания загрузочного образа эти идентификаторы связываются с теми объектами, на которые они ссылаются.

Хотя «волшебные» операции и неупакованные типы позволяют коду Java из Jikes RVM обращаться к памяти по аналогии с обращением через указатели C или C++, эти указатели обладают сильной типизацией и не могут использоваться вместо ссылок. Сильная типизация позволяет обнаруживать ошибки программиста во время компиляции, обеспечивает хорошую интеграцию IDE и поддержку со стороны инстру-

ментария поиска ошибок – все эти факторы положительно повлияли на разработку Jikes RVM.

Потоковая модель

В Java существует интегрированная поддержка многопоточности. В 1998 году поддержка многопоточности уровня операционной системы недостаточно хорошо подходила для многих приложений Java. В частности, типичные реализации многопоточной модели в операционных системах не масштабировались для поддержки сотен потоков в одном процессе, а операции блокировки (необходимые для реализации синхронизируемых методов Java) оказывались неприемлемо дорогими. Для предотвращения этих затрат некоторые виртуальные машины Java реализовывали многопоточную модель сами, с использованием режима, называемого *зелеными потоками* (*green threads*). В этом режиме JVM сама управляет процессом переключения многочисленных потоков Java между меньшим количеством потоков операционной системы (часто один или по одному на каждый процессор в многопроцессорной системе). Главное преимущество зеленой многопоточности заключается в том, что она позволяет использовать реализации блокировки и планирования потоков, адаптированные к семантике Java и хорошо масштабируемые. Основной недостаток – то, что операционная система не имеет ни малейшего представления о том, что делает JVM, и это может привести к некоторым патологиям быстродействия, особенно если приложение Java интенсивно взаимодействует с системным кодом (который и сам делает предположения о потоковой модели). Тем не менее JVM с зелеными потоками стали отправной точкой для ряда проектов операционных систем на базе Java, которые могли избежать патологий быстродействия, возникающих от организации зеленой многопоточности поверх стандартной операционной системы.

С улучшением производительности процессоров и реализаций операционных систем преимущества управления потоками в JVM сократились, хотя управление операциями блокировки без конкуренции в JVM попрежнему предпочтительно.

Применение Java позволило написать логически стройный потоковый API, позволяющий Jikes RVM использовать разные базовые потоковые модели – например, предоставляемые разными операционными системами. В будущем наличие гибкого интерфейса между потоковыми моделями языка и операционной системы может позволить Jikes RVM адаптироваться к новым потребностям программистов (например, поддерживать тысячи программных потоков).

Внешний интерфейс

К сожалению, Jikes RVM не всегда удавалось ограничиваться кодом Java. Для выделения страниц памяти неизбежно приходилось использовать функции операционной системы. Библиотека классов обеспечивает взаимодействие кода Java с существующими библиотеками (например, библиотеками управления окнами). Jikes RVM предоставляет два механизма для обращения к внешнему коду:

JNI (Java Native Interface)

Стандарт JNI позволяет приложениям Java интегрироваться с внешними приложениями, обычно написанными на C. Одна из особенностей JNI – возможность работы с объектами Java и вызова методов Java. При этом необходимо вести список объектов, используемых во внешнем коде, чтобы объекты не были уничтожены в ходе уборки мусора. Это несколько снижает эффективность использования JNI для внешних методов.

SysCall

По способу объявления такие вызовы сходны с внешними методами, но имеют дополнительную аннотацию. Это позволяет повысить эффективность перехода во внешний код и обратно, с тем ограничением, что внешний код не может использовать обратные вызовы к виртуальной машине через интерфейсы JNI. В Jikes RVM механизм SysCall реализуется в виде простого вызова внешнего метода по стандартным конвенциям вызова операционной системы.

Загрузчики классов и рефлексия

Загрузчик классов отвечает за загрузку классов в Jikes RVM и взаимодействие этого процесса с объектной моделью. Рефлексия позволяет приложению запрашивать информацию о типах объектов и даже выполнять вызовы методов объектов, типы которых неизвестны на момент компиляции. Она реализуется с использованием таких объектов, как *java.lang.Class*, или объектов из пакета *java.lang.reflect*, который представляют собой обертки API для функций среды времени выполнения Jikes RVM.

Так как некоторые оптимизации зависят от иерархии классов, в загрузчике классов существуют важные точки принятия решений (runtime hooks), которые могут инициировать перекомпиляцию, если выбранные ранее предположения оказались недействительными. Данная возможность более подробно обсуждалась ранее в разделе «Замена в стеке».

Уборка мусора

MMTk (The Memory Management Toolkit) – инструментарий для построения высокопроизводительных реализаций управления памятью (Blackburn et al., May 2004). За последние пять лет MMTk стала одним из интенсивно используемых компонентов базовой инфраструктуры в исследовательском сообществе, занимающемся уборкой мусора. Высокая портируемость и возможность настройки архитектуры стали ключом к ее успеху. Кроме использования в системе управления памятью Jikes RVM, она также была портирована в среды времени выполнения других языков (например, Rotor). В этом разделе мы лишь в общих чертах затронем некоторые ключевые архитектурные концепции MMTk. За общими сведениями об алгоритмах и концепциях уборки мусора лучше всего обращаться к книге «Garbage Collection: Algorithms for Automatic Dynamic Memory Management» (Jones and Lins, 1996).

MMTk базируется на том же метапринципе, что и Jikes RVM в целом: активное применение агрессивного оптимизирующего компилятора позволяет записывать высокопроизводительный код в высокоуровневом, расширяемом и объектно-ориентированном стиле. Этот принцип получил наибольшее развитие в MMTk, где даже самые критичные по производительности операции (такие как ускоренное (fast path) создание объектов или барьеры записи) четко выражаются в виде хорошо структурированного, объектно-ориентированного кода Java. На первый взгляд, этот стиль полностью несовместим с высокой производительностью (создание даже одного объекта требует десятков виртуальных вызовов на уровне исходного кода и довольно сложных числовых операций).

Тем не менее при рекурсивной подстановке и оптимизации процесса создания объекта оптимизирующим компилятором генерируется компактная последовательность кодов (порядка 10 машинных команд для выделения памяти объекта и инициализации его заголовка), идентичная той, которая может быть реализована ручным программированием ускоренной последовательности на ассемблере.

В MMTk память делится на *пространства* (Spaces), которые представляют собой области виртуальной памяти (возможно, несмежные). MMTk содержит разные реализации пространств, каждая из которых воплощает конкретный примитивный механизм выделения и освобождения фрагментов виртуальной памяти. Например, реализация CopySpace использует выделение непрерывной (bump pointer) памяти, и освобождение посредством копирования всех «живых» объектов в другое пространство; MarkSweepSpace использует списки свободной памяти для хранения информации обо всех свободных блоках, а также поддер-

живает уничтожение непомеченных («мертвых») объектов с возвращением их в соответствующий список.

В терминологии MMTk *планом* (Plan) называется то, что мы обычно представляем себе как алгоритм уборки мусора. Планы определяются объединением одного и более экземпляров пространств разными способами. Например, план MMTk GenMS реализует относительно прямолинейный алгоритм пометки/освобождения с делением на поколения; пространство CopySpace используется для реализации области «молодых» объектов, а пространство MarkSweepSpace – для реализации области «зрелых» объектов. Кроме пространств, используемых для управления кучей Java пользовательского уровня, все планы MMTk также включают несколько пространств для памяти уровня виртуальной машины. В частности, специальные пространства используются для хранения JIT-сгенерированного кода, загрузочного образа Jikes RVM и низкоуровневых объектов реализации виртуальной машины (таких как TIB). Всего существует 15 планов (т. е. разных алгоритмов уборки мусора), заранее определенных и распространяемых в MMTk версии 3.0. Научным сообществом MMTk разработан и описан в академической литературе ряд других планов.

MMTk взаимодействует со своей управляющей системой времени выполнения по двум интерфейсам, определяющим, какие аспекты MMTk API предоставляет виртуальной машине и какие сервисные функции рассчитывает получить от нее. Чтобы сохранить портируемость MMTk для разных виртуальных машин, процесс сборки жестко следит за соблюдением этих интерфейсов, отдельно компилируя MMTk с фиктивной реализацией интерфейса VM. Возможно, самым сложным фрагментом этих интерфейсов является часть, при помощи которой MMTk и Jikes RVM совместно определяют *корни* (*roots*), используемые при уборке мусора. В большинстве планов MMTk представлены отслеживающие (tracing) уборщики. В них уборщик мусора начинает с набора корневых объектов (обычно глобальных переменных программы и ссылок на кадры стека потока) и определяет транзитивное замыкание их ссылочных полей для поиска всех доступных объектов. Доступные объекты считаются «живыми» и не уничтожаются в ходе уборки. Любые объекты, не входящие в транзитивное замыкание, считаются «мертвыми»; они могут быть безопасно уничтожены уборщиком мусора и использованы для удовлетворения будущих запросов памяти. В Jikes RVM корневыми объектами уборки мусора являются регистры, стек, JTOC и загрузочный образ. Ссылки в загрузочном изображении определяются по корневой карте – сжатому представлению всех смещений загрузочного образа, содержащих ссылки. Для определения ссылок в стеке и регистрах метод и конкретная позиция в нем оп-

ределялись по тому же принципу, который использовался для модели исключений (см. «Модель исключений» ранее в этой главе). MMTk получает итератор и использует его для обработки регистров и стека для получения ссылок.

Интеграция с Jikes RVM

Jikes RVM интегрируется с MMTk в ходе исходного создания представлений объектов, предоставляя итераторы для обработки ссылок, создания объектов и реализации барьеров. Создание объектов и барьеры могут влиять на быстродействие; так как инфраструктура MMTk написана на Java, связанный с ней код может быть напрямую связан с компилируемым кодом в целях эффективности. Барьеры необходимы в схемах уборки мусора по различным причинам. Например, барьеры записи позволяют выявить использование объектов, которые могут копироваться параллельно с выполняемым уборщиком мусора, а барьеры записи используются уборщиками с поддержкой поколений, где запись в старый объект означает, что старый объект должен рассматриваться как корневой для освобождения молодых поколений.

Итоги

MMTk предоставляет мощный и популярный набор схем уборки мусора. Удобство интеграции различных модулей Jikes RVM и MMTk упрощает разработку и сокращает непроизводительные затраты, поскольку Jikes RVM подставляет части кода MMTk по соображениям быстродействия. Написание алгоритмов уборки мусора на Java позволяет автору реализации уборки мусора не обращать внимания на то, что происходит внутри компиляторов. Интегрированная многопоточность Java означает, что все уборщики мусора выполняются параллельно и интегрируются с моделью времени выполнения. Библиотеки Java помогают определить интерфейс уборщика мусора, а это означает, что написание новых уборщиков мусора в MMTk значительно упростилось.

Выводы

Jikes RVM – успешная исследовательская виртуальная машина, обеспечивающая производительность, близкую к идеальной; она гибка и легко расширяется. Тот факт, что она написана на языке, поддерживаемом средой времени выполнения, обеспечивает плотную интеграцию и возможность повторного использования компонентов. Применение Java способствует формированию понятной кодовой базы, обеспечивает хорошую модульность и возможность применения высококачественного инструментария (в частности, IDE).

Язык Java и реализации JVM продолжают развиваться, и Jikes RVM приходится развиваться параллельно с ними. Одним из интересных этапов развития стал язык программирования X10, который решает проблемы разработки приложений для многоядерных систем посредством предоставления гарантий потоковой безопасности (по аналогии с тем, как уборка мусора предоставляет гарантии безопасности при работе с памятью в Java). Кодовая база Jikes RVM уже предоставляет отличную тестовую площадку для разработки идей X10. Со временем реализации JVM будут создавать новые оптимизации (скажем, новые приемы уборки мусора или подстановки объектов), и возможность подключения таких оптимизаций к Jikes RVM приведет к совершенствованию среды времени выполнения, компиляторов и кодовой базы, а также продемонстрирует силу метацикличности.

Хорошая расширяемость метациклической среды превращает Jikes в отличную платформу для исследований в области многоязыковых виртуальных машин. Расширение также позволяет записывать отдельные аспекты Jikes RVM на других языках программирования.

Библиография

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1985 «*Structure and Interpretation of Computer Programs*». Cambridge, MA: MIT Press.

Aho, Alfred, Ravi Sethi, and Jeffrey Ullman. 1986 «*Compilers, Principles, Techniques, and Tools*». Boston, MA: Addison-Wesley.

Alpern, Bowen, et al. 2005. «The Jikes Research Virtual Machine project: Building an open-source research community». *IBM Systems Journal*, vol. 44, issue 2.

Blackburn, Steve, Perry Cheng, and Kathryn McKinley. 2004 «*Oil and water? High performance garbage collection in Java with MMTk*» (pp. 137–146). International Conference on Software Engineering, Edinburgh, Scotland. ACM, May '04.

Bodik, Rastislav, Rajiv Gupta, and Vivek Sarkar. 2000 «*ABCD: eliminating array-bounds checks on demand*». ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), Vancouver, British Columbia, Canada. ACM '00.

Choi, Jong-Deok, et al. 1999 «*Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs*». ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99), Toulouse, France: ACM, Sept. '99.

Fink, Stephen, Kathleen Knobe, and Vivek Sarkar. 2000 «*Unified Analysis of Array and Object References in Strongly Typed Languages*». Static Analysis Symposium (SAS 2000), Santa Barbara, CA: Springer Verlag.

Ingalls, Daniel, et al. 1997. «Back to the future: the story of Squeak, a practical Smalltalk written in itself». *ACM SIGPLAN Notices*, vol. 13, issue 10: 318–326.

Jones, Richard and Rafael Lins. 1996 «*Garbage Collection: Algorithms for Automatic Dynamic Memory Management*». Hoboken, NJ: John Wiley and Sons.

McCarthy, John, et al. 1962 «*LISP 1.5 Programmer's Manual*». Cambridge, MA: MIT Press.

Piumarta, Ian, and Fabio Riccardi. 1998. «Optimizing direct threaded code by selective inlining». *ACM SIGPLAN Notices*, vol. 33, issue 5: 291–300.

Rogers, Ian, Jisheng Zhao, and Ian Watson. 2008 «*Boot Image Layout For Jikes RVM*». Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '08), Paphos, Cyprus. July '08.

IV

Архитектуры пользовательских приложений

	Принципы и свойства		Структуры
✓	Гибкость	✓	Модуль
	Концептуальная целостность		Зависимость
✓	Возможность независимого изменения		Обработка
	Автоматическое распространение		Доступ к данным
	Удобство построения		
✓	Адаптация к росту		
✓	Сопротивление энтропии		

11

GNU Emacs: сила ползучей функциональности

Джим Блэнди

Я использую Emacs, который можно назвать термоядерным редактором. Его создал Ричард Столмен; этого вполне достаточно. Он был написан на Lisp, единственном действительно красивом компьютерном языке. Он колоссален, но при этом работает только с простыми текстовыми файлами в формате ASCII, то есть не поддерживает ни шрифтов, ни жирного выделения, ни подчеркивания... Если вы профессиональный писатель – то есть человек, которому платят за то, чтобы вы позаботились о форматировании и печати своих текстов, – Emacs затмевает все остальные редакторы примерно так же, как свет солнца затмевает звезды. Он не просто больше и ярче; на его фоне все остальное попросту меркнет.

Нил Стивенсон

Ни один текстовый редактор не вызывает столько споров, как GNU Emacs. Его сторонники клянутся, что ни один редактор даже отдаленно не сравнится с ним, и упорно не желают поддаваться соблазну более современных альтернатив. Противники называют его запутанным, сложным и устаревшим по сравнению с более широко применяемыми

средами разработки – такими, как Microsoft Visual Studio. Впрочем, даже самые верные поклонники Emacs объясняют спазмы в запястьях извращенным набором клавиатурных команд.

Эмоциональные реакции по поводу Emacs отчасти объясняются его грандиозностью. Текущая версия исходных кодов Emacs состоит из 1,1 миллиона строк программного кода, написанного на собственном языке программирования Emacs – Emacs Lisp. В эту базу включен код, упрощающий редактирование программ на C, Python и других языках, что вполне естественно для текстового редактора, предназначенного для программистов. Но в ней также присутствует код, упрощающий отладку выполняемых программ, совместную работу с другими программистами, чтение электронной почты и новостей, поиск и просмотр каталогов и решение задач символической алгебры.

Если заглянуть внутрь, ситуация выглядит еще более странно. Emacs Lisp не имеет системы объектов, его модульная система сводится к простым схемам построения имен, все фундаментальные операции редактирования текста используют неявные глобальные аргументы, и даже локальные переменные не совсем локальны. Emacs попирает почти все принципы проектирования программ, признанные полезными и нужными. Коду 24 года, он огромен и написан сотнями разных людей. По всем признакам он должен развалиться.

Однако Emacs работает – и работает хорошо. Круг его возможностей расширяется; пользовательский интерфейс обогащается новыми полезными элементами, а сам проект благополучно переживает и серьезные изменения в фундаментальной архитектуре, и редкие выпуски новых версий, и конфликты и расхождения руководства. В чем причина такой живучести? И где лежат ее пределы?

Наконец, чему другие программы могут научиться у Emacs? Когда мы сталкиваемся с новой архитектурой, имеющей общие цели с Emacs, какие вопросы следует задать для оценки ее успеха? В этой главе я поставлю три вопроса, которые, на мой взгляд, отражают самые ценные характеристики архитектуры Emacs.

Emacs в работе

Прежде чем переходить к обсуждению архитектуры, давайте в общих чертах разберемся, что такое Emacs. Это текстовый редактор, с которым вы работаете примерно так же, как с любым другим. При запуске Emacs для файла появляется окно с содержимым данного файла. Пользователь вносит изменения, сохраняет измененное содержимое и выходит из программы. Однако при таком варианте использования Emacs

малоэффективен; он запускается медленнее других популярных текстовых редакторов, а его сильные стороны не проявляются. Когда мне приходится работать в таком режиме, я не использую Emacs.

Emacs обычно запускается один раз, в самом начале сеанса, а затем продолжает работать. В одном сеансе Emacs можно редактировать сколько угодно файлов, сохраняя изменения в ходе работы. Emacs может хранить файлы в памяти без их отображения; таким образом, текущий вывод соответствует той задаче, над которой вы работаете в данный момент, но другие задачи готовы активизироваться с того места, на котором вы их оставили. Опытный пользователь Emacs закрывает файлы только в том случае, если на компьютере не хватает памяти, так что в продолжительном сеансе Emacs могут быть открыты сотни файлов. На рис. 11.1 изображен сеанс Emacs с двумя открытыми фреймами. Левый фрейм разделен на три окна, в которых отображается заставка Emacs, список содержимого каталогов и буфер взаимодействия Lisp. Правый фрейм содержит только одно окно с буфером исходного кода.

На рисунке представлены три важнейших разновидности объектов Emacs: фреймы, окна и буферы.

Рис. 11.1. *Emacs в работе*

Фреймы (*frames*) – так в Emacs называются окна графического интерфейса вашего компьютера. На рисунке изображены два фрейма, прилегающих друг к другу. Если вы используете Emacs на текстовом терминале (например, через *telnet* или подключение *ssh*), то этот терминал тоже является фреймом Emacs. Emacs может одновременно управлять любым количеством графических и терминальных фреймов.

Окна (*windows*) являются подразделами фреймов[1]. Новые окна создаются только делением надвое существующих окон, а при удалении окна занимаемое им пространство возвращается смежным окнам; как следствие окно (или окна) фрейма всегда полностью занимает все его пространство. Всегда существует текущее окно, к которому относятся вводимые с клавиатуры команды. Окна облегчены, т. е. используют небольшое количество ресурсов; в типичном рабочем сценарии они часто появляются и исчезают.

Наконец, *буферы* (*buffers*) содержат редактируемый текст. Emacs хранит текст каждого открытого файла в буфере, но буфер не обязательно связывается с файлом: в нем могут храниться результаты поиска, электронная документация или просто введенный текст, еще не сохраненный ни в каком файле. В каждом окне отображается содержимое некоторого буфера, а один буфер может отображаться в нуле, одном или нескольких окнах.

Важно понимать, что помимо строки состояния в нижней части каждого окна и других аналогичных украшений, в Emacs существует только один способ вывода текста для пользователя: текст помещается в буфер, который затем отображается в окне. Разделы справки, результаты поисков, содержимое каталогов – все эти данные заносятся в буферы с соответствующими именами. На первый взгляд такое решение кажется дешевым трюком реализации, принятым для упрощения внутреннего строения Emacs; однако в действительности оно весьма принципиально, потому что из него следует, что все эти разнообразные виды контента представляют собой обычный редактируемый текст: для перемещения, поиска, организации, усечения и сортировки этих данных применяются те же команды, что и в любом другом текстовом буфере. Выходные данные любой команды могут служить входными данными следующей команды. В этом отношении Emacs резко отлича-

[1] Обратите внимание: то, что в большинстве графических пользовательских интерфейсов называется «окном», в Emacs называется «фреймом», так как в Emacs термин «окно» используется так, как описано выше. Это довольно неудобно, однако терминология Emacs установилась задолго до широкого распространения графических интерфейсов, и разработчики Emacs, видимо, решили не изменять ее.

ется от таких сред, как Microsoft Visual Studio, в которых, например, результаты поиска могут использоваться только способами, предусмотренными разработчиками реализации. Впрочем, пакет Visual Studio в этом отношении не одинок; этот недостаток присущ большинству программ с графическим интерфейсом.

Например, на рисунке в среднем окне левого фрейма изображен перечень файлов каталога. Как и большинство файловых менеджеров, это окно поддерживает клавиатурные команды для копирования, удаления, переименования и сравнения файлов, выбора групп файлов по маске или регулярному выражению и (конечно) просмотра файлов в Emacs. Но, в отличие от большинства файловых менеджеров, сам перечень выводится в виде обычного текста, хранящегося в буфере. К нему могут применяться все стандартные средства поиска Emacs (включая превосходные команды пошагового поиска). Я могу легко вырезать и вставить текст во временный буфер, удалить метаданные слева для получения простого списка имен, воспользоваться поиском по регулярному выражению для исключения файлов, которые меня не интересуют, получить список имен файлов и передать их следующей операции. Когда вы привыкнете работать по такой схеме, обычные файловые менеджеры начинают раздражать: вам кажется, что выводимая информация недоступна, а доступные команды слишком ограничены. В отдельных случаях даже работа в режиме командной строки кажется «блужданием во мраке», потому что в ней труднее увидеть промежуточные результаты своих команд.

Мы подходим к первому вопросу из трех, о которых я говорил в начале этой главы; вопросу, который можно задать о любом пользовательском интерфейсе: насколько просто использовать результаты одной команды в качестве входных данных другой? Связываются ли команды интерфейса друг с другом? Или после вывода результаты оказываются «в тупике»? На мой взгляд, одной из причин, по которым многие программисты сохраняют верность командному процессору Unix (несмотря на обилие логических нестыковок, хилую модель данных и другие слабости), заключается в том, что практически все в ней может быть преобразовано в часть некоего большего сценария. Программу, готовую к использованию в сценариях, написать проще, чем несовместимую, поэтому тупики встречаются редко. Emacs достигает той же цели, но двигаясь по совершенно иному пути.

Архитектура Emacs

Архитектура Emacs следует паттерну «*Модель-Представление-Контроллер*», часто используемому в интерактивных приложениях

(рис. 11.2). В этом паттерне Моделью является базовая структура обрабатываемых данных, Представление выводит данные для пользователя, Контроллер обеспечивает взаимодействие пользователя с представлением (нажатия клавиш, операции с мышью, выбор команд меню и т. д.) и выполняет соответствующие манипуляции с Моделью. В этой главе я буду записывать термины «Модель», «Представление» и «Контроллер» с прописной буквы, когда речь заходит об элементах паттерна. В Emacs Контроллер почти полностью реализован в коде Emacs Lisp. Примитивы Lisp выполняют операции с содержимым буфера (Модель) и структурой окна. Код перерисовки (Представление) обновляет изображение на экране без явных указаний со стороны кода Lisp. Ни реализация буфера, ни код перерисовки не могут изменяться из кода List.

Рис. 11.2. *Паттерн «Модель-Представление-Контроллер» в Emacs*

Модель: буферы

Emacs используется для редактирования текстовых файлов, поэтому центральное место в Модели Emacs занимает *буфер* для хранения текста. Буфер представляет собой обычную неструктурированную строку, в которой разрывы строк обозначаются символами новой строки; это не список логических строк и не дерево узлов, в отличие от модели DOM, используемой в веб-броузерах для представления документов HTML. В Emacs Lisp поддерживаются примитивные операции с буферами, которые выполняют удаление и вставку текста, извлечение фрагментов содержимого буфера в виде строк, а также поиск совпадений по точно заданной подстроке или регулярному выражению. Буфер может содержать символы из различных кодировок, в том числе и тех, которые необходимы для написания сценариев на большинстве азиатских и европейских языков.

Каждый буфер обладает определенным *режимом*, определяющим поведение буфера при редактировании некоторой разновидности текста. В Emacs предусмотрены режимы для редактирования кода C, текста XML и сотен других видов текста. На уровне Lisp режимы используют *привязки локальных клавиш* для определения команд, специфических для данного режима, и *привязки локальных переменных* для поддержания состояния, относящегося к данному буферу.

В совокупности эти особенности делают режим буфера близким аналогом класса объекта: режим определяет, какие команды доступны для буфера, и предоставляет переменные, от значений которых зависят реализации команд Lisp.

Для каждого буфера примитивы редактирования текста ведут *журнал отмены*, в котором содержится информация, достаточная для отмены последствий команд. В журнале отмены сохраняются границы между командами пользователя; это позволяет отменить команду, выполняющую несколько примитивных операций, как единое целое.

При выполнении операций с буфером код Emacs Lisp может использовать целые числа для определения позиций символов в тексте буфера. Это простой способ, однако операции вставки и удаления перед фрагментом текста приводят к частому изменению числовой позиции. Для отслеживания позиции в буфере в процессе изменения его содержимого код Lisp может создавать *маркеры*, перемещаемые вместе с текстом, с которым они связаны. Любая примитивная операция, которая получает целочисленную позицию в буфере, также может получать маркер.

Код Lisp может присоединять к символам буфера *текстовые свойства*. Текстовое свойство состоит из имени и значения; оба компонента

могут быть произвольными объектами Lisp. На логическом уровне свойства символов независимы, но представление текстовых свойств Emacs обеспечивает эффективное хранение серий последовательных символов с идентичными текстовыми свойствами, а Emacs Lisp включает примитивы для быстрого поиска позиций, в которых текстовые свойства изменяются, так что фактически текстовые свойства используются для пометки диапазонов текста. Текстовые свойства определяют, каким образом Emacs следует выводить текст и как реагировать на выполняемые с ним действия мыши. Текстовые свойства могут даже определять специальные клавиатурные команды, доступные для пользователя только тогда, когда курсор находится над указанным текстом. В журнале отмены буфера, наряду с изменениями самого текста, фиксируются изменения его текстовых свойств. Строки Emacs Lisp тоже могут обладать текстовыми свойствами; при извлечении текста из буфера в строковом формате и вставке строки в другой буфер сохраняются свойства текста.

Оверлей (overlay) представляет собой смежный блок текста в буфере. Начальная и конечная позиции оверлея перемешаются вместе с текстом, по аналогии с маркерами. Как и текстовые свойства, оверлеи могут влиять на вывод и чувствительность мыши у того текста, к которому они относятся, однако, в отличие от текстовых свойств, оверлеи не являются частью текста: они не связываются со строками, а изменения граничных точек оверлея не сохраняются в журнале отмены.

Представление: подсистема перерисовки

По мере того как пользователь вводит текст и выполняет операции с окнами, подсистема перерисовки Emacs обеспечивает своевременное обновление изображения. Перерисовка Emacs обладает двумя важными характеристиками:

Emacs обновляет изображение автоматически

Коду Lisp не нужно указывать, как должно обновляться изображение. Он выполняет необходимые операции с текстом буферов, свойствами, оверлеями и конфигурацией окон, не обращая внимания на то, какие части изображения при этом потеряют актуальность. Когда наступит подходящий момент, код перерисовки Emacs проанализирует накопленные изменения и найдет эффективный набор операций вывода, которые приведут изображение в соответствие с новым состоянием Модели. Избавляя код Lisp от ответственности за управление изображением, Emacs значительно упрощает задачу написания расширений.

Emacs обновляет изображение только во время ожидания пользовательского ввода

В одной команде может быть задействовано любое количество примитивных операций с буферами, воздействующих на произвольные части буфера (или, возможно, на разные буферы). Аналогичным образом пользователь может активизировать клавиатурный макрос, выполняющий длинную серию предварительно записанных команд. В таких случаях вместо отображения промежуточных состояний буфера (сопровождаемого раздражающим мерцанием) Emacs откладывает перерисовку до того момента, когда он будет ожидать получения данных от пользователя; в этот момент изображение выводится сразу для конечного состояния. Это избавляет Lisp-программистов от искушения оптимизировать работу с буферами или переставлять оконные примитивы для повышения плавности вывода. Простейший код будет хорошо отображаться.

Автоматическое и эффективное обновление экрана, отражающее последствия применения примитивов редактирования, кардинально упрощает работу авторов, работающих с Lisp. Такое поведение было принято в ряде других систем; в частности, код JavaScript, встроенный в веб-страницы, может изменять содержимое страниц, а броузер автоматически вносит соответствующие изменения в изображение. Однако на удивление многие системы, вроде бы рассчитанные на расширение, требуют тщательного взаимодействия между кодом расширения и кодом обновления изображения, перекладывая это бремя на программиста.

Во времена появления Emacs использовались терминалы, подключенные к компьютеру по нескольким последовательным линиям или модемам. Подключения часто не отличались быстротой, поэтому умение текстового редактора находить оптимальные серии графических операций для обновления экрана серьезно влияло на удобство работы пользователя. Разработчики Emacs приложили немалые усилия, чтобы преуспеть в этом отношении; была разработана иерархия стратегий обновления – от простых, но ограниченных, до более совершенных, но сложных. Последние использовали те же приемы динамического программирования, что и программы сравнения файлов (такие как *diff*), для определения наименьшего набора операций, достаточного для преобразования старого экрана в новый. Хотя в наше время Emacs продолжает использовать эти алгоритмы для минимизации работы по перерисовке, большая часть усилий тратится впустую, поскольку на современных процессорах, каналах связи и экранах идеально работают и более простые алгоритмы.

Контроллер: Emacs Lisp

«Сердцем» Emacs является реализация собственного диалекта Lisp. В принятой в Emacs реализации паттерна «Модель-Представление-Контроллер» код Lisp занимает доминирующее положение в Контроллере: почти каждая команда, активизируемая комбинацией клавиш, из меню или по имени, соответствует некоторой функции Lisp. Именно Emacs Lisp позволил Emacs успешно адаптироваться к широкому спектру функций, предоставляемых современными реализациями Emacs.

Краткий курс Lisp

У новичков в области Lisp нередко возникают трудности с чтением кода. Отчасти это обусловлено тем, что по количеству синтаксических конструкций Lisp уступает многим языкам, однако повышение нагрузки на эти конструкции позволяет реализовать даже более богатый набор возможностей. Чтобы дать читателю некоторое представление о них, рассмотрим следующие правила преобразования кода Python в Lisp:

1. Управляющие конструкции записываются по аналогии с вызовами функций. Например, `while x*y < z: q()` превращается в `while (x*y < z, q())`. Изменился только синтаксис; перед вами все тот же цикл `while`.
2. Инфиксные операторы записываются так, как если бы они были вызовами функций с экзотическими именами. Выражение `x*y < z` принимает вид `<(*(x, y), z)`. Скобки, используемые только для группировки (в отличие от скобок, в которые заключаются аргументы функций), становятся лишними; удалите их.
3. Теперь все, включая управляющие конструкции и примитивные операции, выглядит как вызовы функций. Переместите открывающую скобку каждого «вызова» в точку перед именем функции и уберите запятые. Например, вызов `f(x, y, z)` превращается в `(f x y z)`, а `<(*(x, y), z)` из предыдущего примера – в `(< (* x y) z)`.

Таким образом, после применения всех трех правил код Python `x*y < z: q()` преобразуется в код Lisp `(while (< (* x y) z) (q))`. Это синтаксически правильное выражение Emacs Lisp имеет тот же смысл, что и исходное выражение Python.

Разумеется, это отнюдь не все изменения, однако вы получили представление о сути синтаксиса Lisp. Большая часть Lisp представляет собой простую номенклатуру функций (например, `<` и `*`) и управляющих структур (например, `while`), которые используются так, как показано выше.

А вот как выглядит определение интерактивной команды Emacs для подсчета слов в текущем выделенном блоке текста. Если я поясню, что `"\\<"` – регулярное выражение Emacs, совпадающее с началом слова, вы, вероятно, поймете, как работает эта функция:

```
(defun count-region-words (start end)
  "Count the words in the selected region of text."
  (interactive "r")
  (save-excursion
    (let ((count 0))
      (goto-char start)
      (while (re-search-forward "\\<" end t)
        (setq count (+ count 1))
        (forward-char 1))
      (message "Region has %d words." count))))
```

Пожалуй, кто-то сочтет, что Lisp дает читателям кода слишком мало визуальной информации о том, что же в действительности происходит в коде, и этот невыразительный синтаксис следовало бы заменить другим, в котором более четко различаются примитивные операции, вызовы функций, управляющие конструкции и т. д. Однако некоторые важнейшие возможности Lisp (на которых я сейчас останавливаться не стану) принципиально зависят от единообразия синтаксиса; многочисленные попытки уйти от него успехом не увенчались. По мере накопления опыта многие программисты Lisp начинают считать, что этот синтаксис – не слишком дорогая цена за мощь и гибкость языка.

В Emacs команда представляет собой обычную функцию Emacs Lisp, помеченную программистом (при помощи краткой аннотации в начале определения) как предназначенную для интерактивного использования. Имя команды, вводимое пользователем после комбинации клавиш Meta+x, совпадает с именем функции Lisp. *Командные ассоциации* связывают комбинации клавиш, операции и пункты меню с командами. Базовый код Emacs преобразует нажатия клавиш и операции

с мышью в команды и передает управление соответствующей команде. В сочетании с автоматическим управлением перерисовкой Представления Emacs процесс обработки командных ассоциаций означает, что код Lisp почти никогда не обрабатывает события в цикле событий – хорошее известие для многих программистов, занимающихся разработкой пользовательских интерфейсов.

Emacs и его реализация Lisp обладают некоторыми важными характеристиками:

- Emacs Lisp не отягощается лишними формальностями. Мелкие настройки и расширения реализуются предельно просто: в файл *.emacs*, находящийся в домашнем каталоге, добавляются однострочные выражения, которые Emacs автоматически загружает при запуске. Эти выражения могут загружать существующие пакеты кода Lisp, задавать значения переменных, влияющих на поведение Emacs, переопределять комбинации клавиш и т. д. Полезную команду (вместе с электронной документацией) можно написать всего за десяток строк (пример полного определения команды Emacs приведен в разделе «Краткий курс Lisp»).
- Emacs Lisp интерактивен. Пользователь может вводить определения и выражения в буфере Emacs и немедленно обрабатывать их. К определению можно вернуться и заново обработать его; это не потребует повторной компиляции или перезапуска Emacs. Фактически Emacs представляет собой интегрированную среду разработки программ, написанных на Emacs Lisp.
- Программы, написанные вами на Emacs Lisp, обладают равными правами с кодом Emacs. Любая нетривиальная функция редактирования Emacs написана на Lisp, так что все примитивы и библиотеки Emacs доступны в вашем коде Lisp. Буферы, окна и другие объекты, связанные с редактированием, представлены в коде Lisp обычными значениями. Вы можете передавать их в параметрах и возвращать из функций, сохранять в своих структурах данных и т. д. Хотя компоненты Модели и Представления Emacs жестко закодированы, Emacs не претендует на какие-то особые привилегии в Контроллере.
- Emacs Lisp – полноценный язык программирования. На нем вполне удобно писать достаточно большие программы (сотни тысяч строк).
- Emacs Lisp безопасен. Ошибка в коде может привести к выдаче сообщения, но не к сбою Emacs. Возможные повреждения от ошибок также могут ограничиваться другими способами: например, встроенные журналы отмены буферов позволяют отменить многие непредусмотренные последствия. Это обстоятельство делает разработку на Lisp более приятной и поощряет эксперименты.

- Код Emacs Lisp легко документируется. Определение функции может включать *doc-строку* – текст, объясняющий назначение и основные принципы использования функции. Почти каждая функция Emacs содержит doc-строку, а это означает, что для получения справки пользователю достаточно ввести максимум одну команду. При выводе doc-строки функции Emacs включает гиперссылку на исходный код функции, упрощая просмотр кода Lisp (естественно, doc-строки не подходят для больших пакетов Lisp, в которые обычно включается руководство с более традиционной структурой).
- В Emacs Lisp отсутствует система модулей. Вместо этого специальные правила построения имен предотвращают конфликты разнородных пакетов при загрузке в одном сеансе Emacs, так что пользователи могут обмениваться пакетами кода Emacs Lisp без координации или одобрения разработчиков Emacs. Кроме того, данное решение означает, что любая функция пакета видна в других пакетах. Если код функции достаточно ценен, другие пакеты могут использовать ее, попадая в зависимость от ее поведения.

  Как ни странно, это создает меньше проблем, чем можно было бы предположить. Может, пакеты Emacs Lisp в основном независимы? Но из примерно 1100 файлов Lisp, входящих в стандартную поставку Emacs, свыше 500 содержат функции, используемые в других пакетах. Я полагаю, что разработчики, занимающиеся сопровождением пакетов из поставки Emacs, следят за сохранением совместимости с другими пакетами, а в их сообществе поддерживаются достаточно тесные связи для согласования несовместимых изменений перед их внесением. Пакеты, не включенные в Emacs, при этом просто перестают работать, а у разработчиков появляется стимул включить свои пакеты в Emacs и присоединиться к «внутреннему кругу».

Ползучая функциональность

Ползучая функциональность[1] (creeping featurism) Emacs является прямым следствием его архитектуры. Жизненный цикл типичной функциональной возможности выглядит примерно так:

1. Когда вы впервые замечаете возможность, которую было бы приятно иметь в Emacs, ничто не мешает вам попытаться реализовать ее; как уже было сказано, бюрократические ограничения в Emacs практически отсутствуют. Emacs предоставляет удобную интерактивную

[1] *http://jarf.ru/wiki/Creeping_featurism – Примеч. перев.*

среду для программирования на Lisp. Простая модель буфера и автоматическая перерисовка позволяют сосредоточиться на выполняемой задаче.

2. Когда у вас появляется работоспособное определение команды, вы можете поместить его в свой файл *.emacs*, чтобы оно оставалось постоянно доступным. А если команда используется часто, в файл следует включить код для ее привязки к клавишам.
3. Со временем то, что когда-то было простой командой, может превратиться в целое семейство взаимодействующих команд. На этой стадии команды собираются в пакет, который вы можете передать своим друзьям.
4. Наконец, самые популярные пакеты включаются в стандартную поставку Emacs, и ее стандартный набор функциональных возможностей расширяется.

Похожие процессы действуют и в существующей кодовой базе. По мере того как вы пишете собственные команды, код Emacs становится для вас куда более понятным. Заметив потенциальное улучшение существующей команды, вы можете открыть ее исходный код (из текста справки, как упоминалось выше) и попытаться реализовать свои улучшения. Функции Emacs Lisp могут переопределяться «на ходу», а это упрощает возможные эксперименты. Если ваша идея сработала, вы заносите измененное определение в свой файл *.emacs* для личного пользования и публикуете исправление для включения в официальный исходный код для всех остальных.

Конечно, действительно оригинальные идеи встречаются редко. Многие опытные пользователи часто придумывают некое усовершенствование Emacs, обращаются к документации и обнаруживают, что кто-то другой уже реализовал его. Пользовательская база Emacs, дружащая с Lisp, занимается совершенствованием и настройкой Lisp в течение 20 лет; скорее всего, многие люди уже сталкивались с вашей проблемой, а кто-то мог даже что-то сделать для ее решения.

Но на какой бы основе ни развивался Emacs – с добавлением новых пакетов или с внедрением исправлений, предложенных пользователями, – его развитие идет «из народа» и отражает интересы пользователей: функциональные возможности, от очевидных и до самых экзотических, существуют потому, что кто-то написал их, а другие сочли их полезными. Роль кураторов Emacs, помимо устранения ошибок, принятия исправлений и добавления новых полезных примитивов, в первую очередь заключается в выборе самых популярных и качественно написанных пакетов, уже используемых сообществом, для их включения в официальный исходный код.

Если бы этим все исчерпывалось, «ползучая функциональность» не создавала бы особых проблем. Однако у нее имеются два неприятных побочных эффекта: пользовательский интерфейс программы становится слишком сложным, а сопровождение самой программы усложняется. Если борьба с первым недостатком ведется с переменным успехом, то второй обходится достаточно эффективно.

Ползучая функциональность и сложность пользовательского интерфейса

Сложность пользовательского интерфейса приложения можно оценивать по двум показателям: сложности Модели, с которой работает пользователь, и сложности набора команд для работы с этой моделью.

Насколько сложна Модель?

Сколько должен узнать пользователь, прежде чем он придет в полную уверенность, что Модель приложения пришла в нужное состояние? Нет ли каких-нибудь скрытых или малопонятных состояний, влияющих на смысл Модели?

Документы Microsoft Word имеют сложную модель. Например, Word может автоматически нумеровать разделы и подразделы документа, обновлять нумерацию при вставке и удалении фрагментов, а также обновлять ссылки на определенные разделы текста. Но чтобы эта функция работала так, как ожидалось, необходимо глубокое понимание стилевых таблиц Word. Без него можно легко допустить ошибку, которая не оказывает видимого влияния на содержимое документа, но мешает нормальной работе нумерации (другой пример – спросите специалиста из службы поддержки об «автоматическом обновлении стилей» в Word версии 2003).

В Emacs эта проблема решается просто: редактор не поддерживает стилевых таблиц, автоматической нумерации разделов, колонтитулов, таблиц и других возможностей, типичных для современных систем форматирования текста. Emacs – чисто текстовый редактор, а буфер содержит обычную последовательность символов. Практически в любых обстоятельствах все важные аспекты состояния буфера видны с первого взгляда. В результате пользователи обычно хорошо понимают, что и почему Emacs сделал с содержимым их файлов.

Насколько сложен набор команд?

Насколько легко получить информацию о действиях, актуальных и полезных в конкретном состоянии? Насколько легко получить информацию о возможностях, которые еще не использовались? В этом отноше-

нии пользовательский интерфейс Emacs весьма сложен. Только что запущенный сеанс Emacs без дополнительных настроек распознает свыше 2400 команд и 700 комбинаций клавиш, а в ходе сеанса обычно загружается множество дополнений.

К счастью, новичку не обязательно осваивать сразу все команды Emacs – как и пользователю UNIX не обязательно изучать все команды командного интерпретатора. Новичок вполне может работать с Emacs как с обычным редактором, обладающим графическим интерфейсом: выделять текст мышью, перемещать курсор при помощи клавиш со стрелками, загружать и сохранять файлы командами меню. Неиспользуемые команды не скрывают основную функциональность.

Однако в таком режиме использования Emacs не лучше любого другого редактора. Чтобы эффективно работать с Emacs, пользователь должен читать руководство и электронную документацию и уметь быстро ориентироваться в этих ресурсах. Такие функции, как *grep* и буферы компиляции, интерактивная отладка и индексирование исходного кода, отличают Emacs от конкурентов, но проявляются только по явному запросу – которого не будет, если вы не знаете об их существовании.

Для упрощения подобных исследований в Emacs имеется семейство команд *apropos*. Эти команды получают строку или регулярное выражение и выводят список команд и переменных настройки с подходящими именами или строками документации. Команды *apropos* не заменяют чтения документации, но они достаточно эффективны, когда вы в общих чертах представляете себе, что именно ищете.

В том, что касается сложности такого рода, пользовательскому интерфейсу Emacs присущи многие характеристики интерфейсов командной строки: в них доступны многие команды, пользователю не обязательно знать их все (или хотя бы большую часть), а освоение новой функциональности потребует сознательных усилий.

К счастью, сообществу Emacs удается эффективно устанавливать единые правила, которым должны подчиняться команды, поэтому во многих пакетах соблюдается логическая последовательность. Например, почти все команды Emacs работают в немодальном режиме: стандартные команды перемещения, поиска, переключения буферов, размещения окон и т. д. доступны всегда, поэтому вам не придется беспокоиться о том, как «выйти» из команды. Или другой пример: многие команды запрашивают значения параметров у пользователя стандартными средствами Emacs, благодаря чему в разных пакетах параметры вводятся по единым правилам.

Ползучая функциональность и сопровождение кода

Разумеется, чем больше объем кода, тем больше усилий требуется для его сопровождения. Когда написанный на Lisp пакет выбирается для включения в стандартную поставку Emacs, группа сопровождения приглашает автора пакета участвовать в его сопровождении. Таким образом, с увеличением количества пакетов численность специалистов по сопровождению тоже растет. Если кто-то откажется от ответственности за пакет (например, из-за нехватки времени или потери интереса), руководство группы сопровождения находит нового добровольца или исключает пакет.

Такое решение основано на том, что Emacs представляет собой набор пакетов, а не единое целое. В определенном смысле динамика сопровождения Emacs больше характерна для платформ (скажем, операционных систем), чем для отдельного приложения. Вместо единой управляющей группы, которая выбирает приоритеты и распределяет усилия, существует сообщество независимых разработчиков. Каждый разработчик преследует собственные цели, после чего их усилия сводятся воедино в процессе выбора и консолидации. В конечном итоге бремя сопровождения всей системы не возлагается на одного конкретного человека.

В этом процессе язык Lisp служит важной границей абстракции. Как и в большинстве популярных интерпретируемых языков, код Emacs Lisp в целом изолируется от подробностей реализации интерпретатора Lisp и базовой архитектуры процессора. Аналогичным образом примитивы редактирования Lisp скрывают реализацию буферов, текстовых свойств и других объектов редактирования; в основном Lisp имеет дело лишь с теми характеристиками, которые разработчики обязуются поддерживать в долгосрочной перспективе. Такой подход позволяет относительно свободно совершенствовать и расширять ядро Emacs, написанное на C, без риска нарушить совместимость с существующими пакетами Lisp. Например, буферы Emacs обрели поддержку текстовых свойств, оверлеев и выбора кодировки практически без нарушения совместимости с кодом, написанным до появления этих новых возможностей.

Две другие архитектуры

Многие приложения поддерживают пользовательские расширения. Интерфейсы расширений присутствуют везде, от систем совместной разработки программных продуктов (например, модули Trac) до программ обработки текстов (Universal Network Objects в Open Office) и систем

управления версиями (расширения Mercurial). В этом разделе Emacs сравнивается с двумя архитектурами, поддерживающими пользовательские расширения.

Eclipse

Хотя большинство людей знает Eclipse в качестве популярной, распространяемой с открытым кодом интегрированной среды разработки для Java и C++, в своей основе Eclipse почти не содержит функциональности. По сути это инфраструктура для подключения модулей, которая обеспечивает простое взаимодействие компонентов, поддерживающих разные аспекты разработки (написание кода Java, отладку выполняемых программ, использование программ управления версиями), легко взаимодействовать друг с другом. Архитектура Eclipse позволяет программистам, располагающим качественной технологией решения одного аспекта задачи, объединить свои усилия для построения унифицированной, полнофункциональной среды разработки.

Eclipse как среда разработки обладает рядом полезных возможностей, отсутствующих в Emacs. Например, модули Java Development Tools предоставляют широкую поддержку рефакторинга и анализа кода. Напротив, Emacs обладает весьма ограниченным представлением о семантической структуре редактируемых программ и не может предложить сравнимого уровня функциональности.

Архитектура Eclipse предельно открыта; практически вся значимая функциональность предоставляется модулями расширения. В системе нет «второсортного» кода; популярные модули расширения строятся на единых интерфейсах, доступных другим. А поскольку Eclipse предоставляет каждому модулю расширения относительно низкоуровневый контроль над вводом и выводом, модуль может сам выбрать Модель, Представление и Контроллер, наилучшим образом отвечающие его целям.

Однако такой подход обладает рядом недостатков:

- Разработка модулей расширения Eclipse небезопасна (если говорить о безопасности в контексте кода Emacs Lisp). Неверно написанный модуль расширения легко приводит к сбою или зависанию Eclipse. В Emacs интерпретатор Lisp следит за тем, чтобы пользователь всегда мог прервать вышедший из-под контроля код Lisp, а четкая граница между кодом Lisp и реализацией Модели защищает данные пользователя от более коварных видов повреждений.
- Интерфейсы, предоставляемые модулями Eclipse друг другу, относительно сложны, поэтому написание модуля расширения Eclipse больше напоминает включение модуля в сложное приложение, не-

жели написание сценария. Разумеется, именно благодаря этим интерфейсам стала возможной вся функциональность Eclipse, однако авторам модулей расширения приходится платить эту цену как за сложные, так и за простые проекты.

- Модуль расширения должен содержать немалый объем шаблонного кода, поэтому в Eclipse был включен модуль расширения, который помогает другим людям создавать модули расширения. Eclipse Plug-in Development Environment может генерировать заготовки кода, записывать стандартные конфигурационные файлы в формате XML и манифесты, создавать и уничтожать тестовые среды. Автоматическое генерирование шаблонного кода «мастером» помогает программисту начать работу, но не сокращает сложности нижележащих интерфейсов.

Как следствие, модули расширения Eclipse не отличаются особой гибкостью. Они плохо подходят для задач автоматизации «на скорую руку» и не особенно удобны для повседневных экспериментов с пользовательским интерфейсом.

Мы подходим ко второму вопросу из трех, обещанных мной в начале главы, – вопросу, который относится к любому модульному механизму: какие интерфейсы доступны для используемых модулей? Достаточно ли они просты для быстрой разработки, характерной для сценарных языков? Может ли разработчик модуля работать на более высоком уровне абстракции, близком к предметной области задачи? И как данные приложения защищены от ошибок в коде модуля расширения?

Firefox

Современное поколение веб-приложений (Google Mail, Facebook и т. д.) интенсивно применяет такие технологии, как *динамический HTML* и *AJAX*, для повышения качества пользовательских взаимодействий. Веб-страницы таких приложений содержат код JavaScript, который локально реагирует на действия пользователя, а затем связывается с основными серверами по мере надобности. Для перемещения и изменения содержимого страницы код JavaScript использует стандартный интерфейс *DOM* (Document Object Model), а дополнительные стандарты определяют внешний вид страницы. Все современные броузеры в той или иной степени реализуют эти стандарты.

Конечно, броузер – не текстовый редактор, однако между архитектурой Emacs и архитектурой броузера существует разительное сходство:

- Emacs Lisp и JavaScript имеют мало общего на синтаксическом уровне, однако в их семантике прослеживается ряд сходных аспектов: как и Emacs Lisp, язык JavaScript является интерпретируемым,

динамичным и безопасным. В обоих языках используется уборка мусора.

- Как и при программировании на Emacs Lisp, работу с JavaScript удобно начать с небольшого фрагмента, улучшающего второстепенный аспект поведения страницы, а затем постепенно превращать его во что-то более значительное. Начать работу с языком на простейшем уровне несложно, но язык хорошо масштабируется и для более крупных задач.
- Как и в Emacs, управление перерисовкой осуществляется автоматически. Код JavaScript просто изменяет дерево узлов, представляющих содержимое веб-страницы, а броузер обеспечивает обновление изображения в соответствии с изменениями модели.
- Как и в Emacs, процесс передачи событий ввода коду JavaScript проходит под управлением броузера. Firefox решает, какому элементу веб-страницы адресовано событие, находит соответствующий обработчик и активизирует его.

Однако Firefox продвигает идеи, заложенные в основу современных веб-приложений, чуть дальше: собственный пользовательский интерфейс Firefox реализуется тем же базовым кодом, который отображает веб-страницы и обеспечивает их взаимодействие. Набор пакетов *оформления* (*chrome*) описывает структуру и стиль интерфейса, а также включает код JavaScript, обеспечивающий его работу[1]. Такая архитектура позволяет независимым разработчикам создавать дополнения, расширяющие пользовательский интерфейс Firefox новым оформлением. Более того, разработчики могут полностью заменить стандартное оформление Firefox и кардинально переработать весь пользовательский интерфейс – например, адаптировать его для мобильных устройств.

Как и модули расширения Eclipse, пакеты оформления Firefox включают значительный объем метаданных. И по аналогии с модулем расширения Eclipse Plug-in Development Environment существует расширение Firefox, упрощающее написание расширений Firefox. Впрочем, прежде чем вы сможете создать свое расширение или исправление Firefox, вам придется проделать изрядный объем работы. Однако благодаря автоматической перерисовке и упрощенному механизму обработки событий Firefox это потребует существенно меньших усилий, чем написание модуля расширения Eclipse.

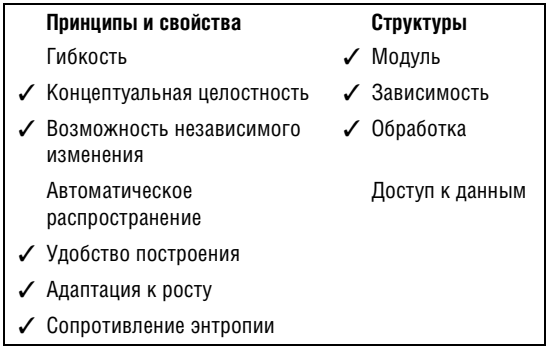
---

[1] Естественно, код JavaScript, используемый в оформлении, может читать и записывать файлы конфигурации, таблицы закладок и обычные файлы – эти привилегии никогда не предоставляются коду, загруженному с веб-страницы.

Разработчики Firefox работают над повышением быстродействия реализации JavaScript. Очевидно, это поможет пользователям, посещающим сайты с интенсивным использованием JavaScript, а также позволит разработчикам Firefox переносить все большую часть кода самого броузера с C++ на JavaScript, гораздо более удобный и гибкий язык для решения этой задачи. В этом смысле архитектура Firefox все сильнее напоминает Emacs с его Контроллером, написанным полностью на Lisp.

Мы подходим к третьему и последнему вопросу, который можно задать о любом языке расширений: является ли этот язык предпочтительным механизмом реализации большинства новых функций приложения? А если нет, какие ограничения препятствуют его применению? Слабости самого языка? Громоздкость его интерфейса с Моделью? Независимо от причины, с этими недостатками, вероятно, сталкиваются и разработчики расширений, в результате чего расширения становятся «второсортными» элементами системы (аналогичные вопросы относятся и к интерфейсам модулей расширения). Как и в случае с Emacs, язык расширений занимает центральное место в архитектуре Firefox – убедительный аргумент в пользу того, что связи языка с приложением были спроектированы правильно.

Я принадлежу к числу ревностных пользователей Emacs. Однако я беспокоюсь о будущем продукта, поэтому меня особенно интересует Firefox, во многих отношениях столь близкий к Emacs: Представление с автоматическим управлением перерисовкой; Контроллер, основанный на интерпретируемом динамическом языке; Модель, которая делает все, что делает Emacs, а также многое другое. Если бы кто-то пожелал оставить позади всю накопленную кодовую базу Emacs Lisp, то после нескольких дней программирования оформления можно было бы создать текстовый редактор с архитектурой, очень близкой к архитектуре Emacs, но обладающий значительно более мощной моделью и находящийся ближе к передовым рубежам современных технологий. Самые ценные уроки, которым нас учит архитектура Emacs, забывать нельзя.



12

Когда базар строит собор

Как ThreadWeaver и Akonadi формировались под влиянием сообщества KDE и как они повлияли на это сообщество

Тиль Адам
Мирко Бём

Введение

KDE – один из крупнейших мировых проектов в области ПО с открытым кодом. За прошедшие 10 лет участники этого крайне разнородного сообщества – студенты, опытные профессионалы, любители, компании, правительственные агентства и т. д. – создали огромное количество программ, предназначенных для решения разнообразных задач, от полноценной настольной графической среды с броузером, пакетом коллективной работы, файловым менеджером, редактором, электронной таблицей и средствами проведения презентаций до узкоспециализированных приложений (например, виртуального планетария). В основе всех перечисленных приложений лежит набор общих библиотек, поддерживаемых коллективными усилиями сообщества. Кроме сообщества разработчиков KDE (основной аудитории) эти библиотеки также ис-

пользуются многими независимыми разработчиками как на коммерческой, так и на некоммерческой основе, для создания тысяч дополнительных приложений и компонентов.

Изначально главной целью проекта KDE было создание интегрированной настольной среды для бесплатных операционных систем Unix, прежде всего GNU/Linux. Однако с тех пор масштаб KDE значительно расширился, и многие программы проекта теперь доступны не только в разных версиях Unix, но и в Microsoft Windows и Mac OS X, а также на встроенных платформах. Отсюда следует, что код, написанный для библиотек KDE, должен работать со многими инструментариями, справляться с особенностями разных платформ, гибко интегрироваться с системными службами с сохранением возможности расширения, осмотрительно и экономно использовать аппаратные ресурсы. Широта круга потенциальных пользователей библиотек также означает, что они должны предоставлять API понятный, практичный и адаптируемый для программистов с разной подготовкой. Программист, привыкший иметь дело с технологиями Microsoft для Windows, будет иметь другие предрассудки, субъективные наклонности, привычки и инструменты, чем программист, занимающийся разработкой встроенных систем с опытом работы на Java, или опытный разработчик для платформы Mac. Цель заключается в том, чтобы все программисты могли работать удобно и продуктивно, чтобы они могли решать насущные задачи, а также (причем кое-кто считает, что это более важно) чтобы их вклад приносил пользу проекту, если они захотят поделиться своими рекомендациями, усовершенствованиями и расширениями.

Это очень гибкая, очень разносторонняя и соревновательная экосистема, в которой самые активные участники заинтересованы в улучшении своих программ и навыков посредством постоянного изучения работ друг друга. Мнения высказываются свободно, причем споры порой бывают весьма ожесточенными. Что-то всегда можно сделать лучше, а код постоянно анализируется очень умными людьми. Студенты, изучающие компьютерные технологии, анализируют реализации на занятиях. Компании выявляют ошибки и публикуют инструкции по безопасности. Новые участники стараются продемонстрировать свое мастерство, улучшая существующий код. Производители оборудования берут настольную среду и втискивают ее в мобильные телефоны. Люди неравнодушны к своей работе и к тому, как, по их мнению, следует подходить к решению этих задач. Происходящее во многих отношениях напоминает пресловутый базар[1]. Но и этой буйной, неуправляемой публике приходится решать те же проблемы, что и более тра-

[1] *http://en.wikipedia.org/wiki/The_Cathedral_and_the_Bazaar*

дициональным организациям, занимающимся сопровождением большого количества библиотек и приложений.

Некоторые проблемы имеют чисто техническую природу. Программам приходится обрабатывать все большие объемы данных, причем сами данные все усложняются. Необходимость взаимодействия с другими программами (бесплатными и коммерческими), используемыми в корпорациях и правительственных учреждениях, приводит к сдвигу отраслевой парадигмы – например, ориентации на архитектуры SOA (Service-Oriented Architecture). Применение для решения важнейших правительственных и корпоративных задач предъявляет жесткие требования к безопасности, а крупномасштабное развертывание требует качественной автоматизации. Новички, дети, взрослые и т. д. обладают совершенно разными потребностями, однако все они считаются равноправными, и им уделяется равное внимание[1]. Как и большая часть отрасли, программные продукты с открытым кодом (и KDE в частности) находят свое место в пользующихся все большей популярностью областях многопоточности и распределенной обработки данных. Пользователи с полным правом требуют более удобных, чистых и красивых интерфейсов; оперативных, хорошо продуманных и не слишком сложных взаимодействий с программами; высокой надежности, стабильности и безопасности данных. Разработчики желают видеть точки расширения, нейтральные по отношению к языкам, хорошее сопровождение API, совместимость на двоичном уровне и уровне исходного кода, чистые процессы миграции и функциональность на уровне коммерческих аналогов, к которым они привыкают в своей повседневной работе.

Но еще более устрашающе выглядят социальные и координационные аспекты крупного проекта в сфере свободного ПО. Общение – ключ к успеху, но на его пути встают многочисленные препятствия: часовые пояса, культурные барьеры, более или менее устойчивые предрассудки и предпочтения, а порой и обычные географические расстояния. Обсуждение, которое укладывается в 15 минут на обычном совещании в офисе, порой требует многодневных споров в списках рассылки – ведь мнения участников группы из другого полушария тоже должны

[1] В этом заключается принципиальное отличие от программ, производимых в коммерческих целях. Например, программы с открытым кодом могут адаптироваться к потребностям слепых, и для этого разработчикам не придется искать оправдания затраченных усилий в виде цифр продаж или ожидаемой рентабельности капиталовложений. Скажем, этот фактор объясняет, почему в KDE реализована поддержка значительно большего количества языков, чем у коммерческих конкурентных продуктов.

быть услышаны. Для сохранения сплоченности группы процесс достижения консенсуса часто оказывается, по крайней мере, не менее важным, чем итоговое решение. Естественно, для достижения благоприятного консенсуса требуется больше терпения, понимания и красноречия. Это не так просто для людей, которые преуспевают в математике и физике, но не любят ходить к парикмахеру, потому что им не нравится общаться с непрограммистами. Вот почему дружественная атмосфера ежегодных конференций Akademy выглядит просто потрясающе (см. следующий раздел). Мы еще никогда не видели, чтобы столь разные люди со всего света – молодые и старые, из враждующих стран, принадлежащие к бесчисленным нациям, порой, словно из разных миров, с разным цветом кожи – забывали о различиях и обсуждали (эмоционально, но рационально) малопонятные тонкости C++.

Кроме технических и личностных аспектов существует и третий фактор, влияющий на судьбу проектов с открытым кодом: структура. В определенном отношении структура неизбежна в группах ПО с открытым кодом. В крупных, хорошо известных проектах особенно серьезно встают проблемы регистрации товарных знаков, получения пожертвований и организации конференций. Время, когда проблемы решаются поездками на выходные и доброй волей домашних, обычно кончается при первом публичном упоминании проекта. В определенном отношении структура (или ее отсутствие) определяет судьбу сообщества. Необходимо принять важное решение – выбрать между корпоративно-скучным и неформально-хаотичным путем развития. Выбор далеко не тривиален, потому что приходится учитывать множество факторов: финансирование или возможность свободно влиять на коллег; стабильность процесса разработки или привлечение самых выдающихся «пещерных людей»[1]; открытость и свобода обмена мнениями или ориентация на впечатляющие технические результаты.

При выходе за пределы, в которых структура остается на уровне минимальной необходимости, возможны разные уровни бюрократии. У одних проектов нет даже совета директоров, а у других имеются свои диктаторы, заявляющие о своей доброжелательности. Существуют примеры успешных и неуспешных сообществ на обоих концах этой оси.

Прежде чем переходить к подробному рассмотрению двух конкретных примеров того, как эти технические, социальные и структурные проблемы решались группами из проекта KDE, будет полезно познакомиться с общей историей и традициями проекта KDE в этих трех об-

[1] Мы используем этот термин с большим почтением. Впрочем, нельзя отрицать, что многим из наших лучших друзей не помешало бы чуть больше солнечного света и более здоровой еды.

ластях. В следующем разделе мы расскажем, что собой представляет KDE сегодня, и как сообщество пришло к нынешнему состоянию дел.

История и структура проекта KDE

Проект KDE (K Desktop Environment) был порожден отчаянием. В те времена, когда оконный менеджер FVWM считался рабочим столом, Xeyes была неотъемлемой программой на экране, а инструментарий Motif соревновался с XForms в области истребления мозговых клеток и снижения сексуальности программистов, проект KDE был основан для достижения революционной цели: объединения мощи Unix с эффектным, приятным для глаза пользовательским интерфейсом. Было решено, что эта цель достижима, потому что норвежская фирма Trolltech собиралась выпустить первую версию своего революционного объектно-ориентированного GUI-инструментария Qt для C++. Предполагалось, что Qt сделает GUI-программирование именно таким, каким оно должно быть: систематичным, объектно-ориентированным, элегантным, простым в изучении, хорошо документированным и эффективным. В 1996 году Маттиас Эттрих (Matthias Ettrich), в то время студент Тюбингенского университета, первым описал важность потенциала Qt для разработки полноценной среды рабочего стола. Идея быстро собрала группу из примерно 30 разработчиков, и с тех пор численность этой группы постоянно растет.

Версия 1.0 KDE была выпущена в 1998 году. С сегодняшней точки зрения ее функциональность выглядит весьма легковесной, но ее следует оценивать по сравнению с основными конкурентами: в то время у Windows 3.1 не было защиты памяти, фирма Apple искала новое ядро, а Sun сметала в мусорное ведро жалкие остатки CDE. Кроме того, шумиха по поводу Linux еще не началась, а движение свободного ПО еще не получило признания у разработчиков.

Даже в процессе завершения KDE 1 разработчики уже начали перерабатывать архитектуру продукта для версии 2.0. В ней отсутствовали некоторые важные элементы: модель компонентов, уровень сетевой абстракции, протокол взаимодействия с рабочим столом, API стилевого оформления интерфейса и многое другое. Только начиная с KDE 2 процессы разработки архитектуры, проектирования и реализации стали относительно рациональными. Технология Corba рассматривалась для реализации компонентной модели, но была отвергнута. В это же время произошли первые изменения в составе участников. Хотя некоторые основные разработчики, участвовавшие на ранних стадиях, ушли, размер активной группы разработки KDE медленно, но верно увеличивался. В 1996 году была основана KDE e.V., организация участ-

ников проекта KDE, а к 1998 году она представляла абсолютное их большинство. С самого начала организация предназначалась для поддержки участников, но не для влияния на техническое руководство KDE. Предполагалось, что путь развития продукта должен быть результатом работы активных разработчиков, а не руководящей команды. Эта идея оказалась одним из важнейших факторов (если не самым важным), вследствие которых проект KDE стал одним из очень немногочисленных проектов с открытым кодом, не попавших под подавляющее влияние финансирующих корпоративных организаций. У KDE не было проблем с получением внешнего финансирования, но многие другие проекты, которые слишком сильно зависели от него, переставали существовать с банкротством или потерей интереса со стороны главного спонсора. По этой причине проект KDE пережил много новомодных течений и поветрий в мире ПО с открытым кодом, не теряя темпа разработки.

В апреле 2002 года появилась версия KDE 3. Так как версия KDE 2 считалась хорошо спроектированной, версия 3 имела эволюционную природу, а ее движение к совершенству заняло шесть лет и пять основных версий. На базе технологий KDE 3 были созданы важные приложения, ставшие стандартными для бесплатных настольных сред: K3B, программа записи дисков; Amarok, один из самых симпатичных музыкальных проигрывателей; Kontact, полный пакет для личных коммуникаций. Интересно заметить, что эти приложения впервые использовали KDE не как одну целевую среду, а как платформу, на базе которой строятся приложения для конечного пользователя. С выходом версии 3 проект KDE начал разделяться на две отдельные ветви: рабочий стол и среду, обычно называемую платформой. Но так как KDE все еще ограничивался рамками X11, не все пользователи увидели это деление. Оно стало следующим шагом в развитии проекта.

В 2004 году группа KDE столкнулась с одним из самых сложных решений в своей истории. Фирма Trolltech готовилась выпустить Qt версии 4.0; новая версия стала очень мощной и сильно отличалась как от предыдущих версий, так и от любых других известных инструментариев. Из-за серьезных изменений в инструментарии переход с Qt 3 на Qt 4 из адаптации превратился в портирование. Вопрос был в том, будет ли KDE 4 результатом прямого портирования KDE 3 в результате перехода с Qt 3 на Qt 4 или же подвергнется капитальной переработке в ходе портирования. У обоих вариантов было много сторонников; абсолютному большинству участников было ясно, что в любом случае потребуется проделать огромный объем работы. Решение было принято в пользу полной переработки KDE. Несмотря на то, что сейчас это решение считается правильным, оно было крайне рискованным, пото-

му что из него следовала необходимость продолжительной параллельной поддержки KDE 3 вплоть до завершения титанической работы по портированию.

На одной новой возможности Qt 4 следует остановиться особо. GPL и схема двойного коммерческого лицензирования, уже использованная Trolltech в версии для X11, была расширена на все целевые платформы, поддерживаемые Qt, – в первую очередь Windows, Mac OS X и встроенные платформы. Таким образом, у KDE 4 появилась возможность выйти за рамки мира UNIX. Рабочий стол UNIX остается «родной территорией» KDE, но приложения, разработанные для KDE, теперь могут выполняться на компьютерах с Windows и Mac OS X. Данная возможность была встречена неоднозначно. Один из аргументов против нее заключался в том, что сообщество свободного ПО будет создавать качественные приложения для этих закрытых платформ, лишая пользователей стимула для перехода на альтернативы с открытым кодом. Другой вопрос звучал так: «Какое нам до них дело?» или более вежливо: «Зачем нам тратить драгоценное время разработки на поддержку коммерческих целевых систем?». Сторонники возражали, что распространение одних и тех же приложений на всех платформах упростит переход пользователей с коммерческих операционных систем на бесплатные и обеспечит возможность постепенной замены ключевых приложений. В конечном итоге решили действовать по традиционной для KDE мантре «решают те, кто работает». Интерес к новым целевым платформам был достаточно высок для привлечения значительного количества участников, так что в конечном итоге не было причин лишать пользователей KDE очевидно привлекательной для многих возможности.

Для достижения платформенной независимости архитектура KDE 4 была переработана, а проект разделился на платформу разработки приложений, построенный на ее основе рабочий стол и сопутствующие приложения. Конечно, граница деления иногда представляется весьма расплывчатой.

Итак, принимая во внимание эту предысторию, мы посмотрим, как в проектах типа KDE пытаются решать проблемы, выходящие за рамки возможностей одного разработчика, пусть даже самого одаренного. Как сберечь коллективные знания и опыт и использовать их в максимальной мере? Как при реализации стратегического мышления в большой, разнородной группе принимать трудные решения, чтобы совместно двигаться к общей цели, не теряя удовольствия от работы? Как воплотить в реальность идею равноправного принятия решений и другие аспекты, обусловившие успех KDE и других проектов свободного ПО? Другими словами, как построить собор, когда все участники

строительства склонны считать себя архитекторами (с основаниями или без), и как выполнять функции архитектора без лишнего пафоса? Начнем с участников, потому что мы убеждены, что сообщества свободного ПО в первую очередь являются социальными, а не техническими структурами. Люди – наш самый ценный и редкий ресурс.

В каждом проекте задействовано множество разных ролей, и не существует авторитетов, принимающих решения в части привлечения людей к проекту. Каждому проекту свободного ПО необходимы представители для внешнего общения, а также множество квалифицированных, практичных, заинтересованных в конечном результате технарей, администраторов, художников, авторов, переводчиков и т. д. Похоже, у всех этих людей есть только одно общее свойство: от них требуется большой энтузиазм, мастерство и самостоятельность. Возможность внести свой вклад в общее дело привлекает многих выдающихся личностей всех возрастов – от студентов до пенсионеров. Они присоединяются к проекту, чтобы стать частью группы, создающей технологии завтрашнего дня, чтобы познакомиться с другими людьми, интересующимися этими технологиями и определяющими пути их развития, а часто еще и для того, чтобы сделать что-то действительно полезное вместо курсовых работ и рефератов, которые никому не нужны.

Хорошо налаженное сообщество свободного ПО является едва ли не самой конкурентоспособной средой из всех возможных. Большинство коммерческих технических групп, которые нам встречались, были гораздо менее свободны, а на их участников постоянно давила политика защиты корпоративных капиталовложений. С разработчиками свободного ПО дело обстоит совершенно иначе. Единственным критерием для включения кода в программный продукт является его качество и возможность сопровождения. В течение какого-то времени посредственный фрагмент кода, написанный заслуженным программистом, может оставаться в исходном коде продукта, но рано или поздно он будет заменен. Код постоянно остается на виду, постоянно анализируется. Даже если реализация хороша, но недостаточно близка к совершенству по мнению других участников, группа программистов возьмется за нее и усовершенствует. Самым сильным стимулом для работы является возможность творить, поэтому разработчики свободного ПО должны постоянно помнить, что их творение уйдет в небытие с появлением следующей версии. Это объясняет, почему код проектов свободного ПО часто превосходит по качеству код коммерческих проектов – участники не могут надеяться на то, что небрежная работа останется незамеченной.

Как ни странно, программистам редко требуется внешняя мотивация в виде признания пользователей или СМИ. Часто они теряют интерес

к своей работе незадолго до выпуска окончательной версии. Похоже, их удовлетворение от работы сродни чувству марафонского бегуна – осознание того, что ты наконец-то пересек финишную черту. Иногда это качество понимают как стеснительность; напротив, оно лишь подчеркивает безразличие к почестям. Из-за этого набора личных качеств в действительно свободных программных проектах почти невозможно назначить приоритеты целям разработки. Искусные программисты руководствуются внутренней мотивацией, они выбирают и назначают свои собственные приоритеты. Работа над программным продуктом все равно приходит к успешному завершению, потому что самые сложные элементы привлекают наибольшее внимание. Эта внутренняя координация вместо поставленных внешних целей идет только на пользу и приводит к желаемым результатам. Выбранный путь нередко отличается от того, который был бы установлен в хорошо управляемом коммерческом проекте, и обычно программисту приходится реже идти на компромиссы, жертвуя качеством ради соблюдения сроков. Если такие компромиссы принимаются регулярно, обычно это свидетельствует о том, что проект свободного ПО находится на пути преобразования в коммерческий, «менее свободный» проект. Как правило, это происходит, когда группа разработчиков создает на базе проекта компанию, спонсор берет под контроль основных участников или сам проект перестает стремиться к новизне и переходит в режим сопровождения.

Большинство структурных и организационных ограничений, устанавливаемых для проекта свободного ПО, встречает сопротивление. Участники присоединяются к проекту KDE по разным причинам, но уж точно не из желания столкнуться с бюрократическими сторонами проекта. Как правило, они отлично понимают и защищают свою свободу выбора. В то же время они признают необходимость минимальных элементов формальной организации. Хорошая группа способна справиться практически с любой технической задачей, но центробежные силы сильнее всего проявляются именно в области формальной структуры. Многие проекты свободного ПО распались из-за решений политического характера, принудительно насаждаемых влиятельными участниками. Определение формальной структуры, которая способна решать возникающие проблемы по мере их появления, но в то же время не препятствует техническому развитию, – один из самых важных (если не самый важный) шагов в истории KDE. Факт формирования очень стабильной структуры, принятой сообществом, бесспорно оказал значительное влияние на долгосрочную стабильность сообщества KDE.

В 1996 году проект KDE основал KDE e.V. – организацию, представляющую участников KDE. e.V., или «eingetragener Verein» – классическая разновидность некоммерческих организаций в Германии, где

в то время проживали и встречались многие участники KDE. Основной причиной для создания KDE e.V. стала необходимость вступления правоспособной организации, представляющей проект, в Free Qt Foundation. Последняя – результат соглашения между Trolltech (создатели Qt) и KDE, обеспечивающего KDE бесплатный доступ к инструментарию Qt. Соглашение гарантировало (и до сих пор гарантирует), что если Trolltech прекратит разрабатывать и публиковать бесплатную версию Qt, то разработка последней опубликованной бесплатной версии может быть продолжена без участия Trolltech. Соглашение играло важную роль, когда бесплатная версия Qt еще не публиковалась на условиях лицензии GPL. В наше время эта проблема решена, а организация в основном занимается нюансами из области авторских прав, соблюдением прав разработчиков и т. д. Организация была важна еще и потому, что многие разработчики не захотели бы тратить время на проект, который мог перейти на коммерческую основу; кроме того, сам факт ее существования показывал, что проект KDE может осуществлять законные права. Позднее организация KDE e.V. зарегистрировала и вступила в права владения товарным знаком KDE, что предполагало целый ряд обязанностей – например, проведение ежегодной крупной конференции KDE.

Организация все еще сохраняет значительное влияние в областях архитектуры и проектирования KDE, хотя она и не обладает правом управления разработкой. Так как считается, что все «заслуженные» участники KDE являются членами организации, мнение KDE e.V. весьма весомо. Также организация занимается проведением ежегодной конференции Akademy – для большинства участников это единственная возможность встретиться и лично обсудить ситуацию. Кроме того, организация занимается сбором средств (прежде всего пожертвований и взносов от спонсоров), что позволяет ей финансировать (или в очень редких случаях не финансировать) различные мероприятия разработчиков – скажем, целевые встречи. Тем не менее, ежегодный бюджет организации на удивление невелик по сравнению с последствиями ее работы. На встречах разработчиков происходит основная координация работы, что становится лишним доводом в пользу членства в организации. И все же большинство задач решается инициативой групп участников, а не заявлениями на генеральной ассамблее.

Конференция Akademy получила широкую известность за пределами сообщества KDE. Именно здесь происходит большая часть координации, требующая личных встреч. Поскольку ни кадры, ни средства не могут напрямую выделяться по любой инициативе разработчиков, многие обсуждения более широкого плана откладываются до конференции. Сейчас вошло в традицию принимать основные решения на

ежегодном собрании, ограничиваясь минимальной координацией во время непосредственной работы. Конференция почти всегда проходит летом. Участники других проектов свободного ПО пользуются этой возможностью координировать свою работу с KDE. В частности, на конференции 2007 года было принято решение о переходе на 6-месячный цикл выпуска версий; эта идея была предложена Марком Шаттлуортом (Marc Shuttleworth) из Ubuntu.

Akademy – единственная всемирная конференция, проводимая KDE. Кроме этой крупной конференции проходит много менее масштабных встреч подгрупп. Такие встречи обычно проводятся чаще, имеют более локальную природу, и на них обсуждаются более узкоспециализированные темы; таким образом, если архитектурные аспекты обсуждаются на конференции Akademy, на встречах подгрупп обсуждаются нюансы проектирования определенных модулей или приложений. Некоторые подгруппы – скажем, разработчики KOffice или Akonadi – обычно встречаются с периодичностью в три месяца.

Процесс координированного анализа высокого и среднего уровня оказался весьма эффективным. Кроме того, он дает разработчикам хорошее понимание целей и возможность дальнейшего планирования. Многие участники считают, что ежегодная конференция способствует их мотивации и повышает эффективность разработки.

Современная организация и структура KDE не является изобретением группы руководителей, которые думали над тем, как должен быть организован проект свободного ПО. Она появилась в результате итеративного поиска подходящей структуры для главных нетехнических целей – сохранения свободы, обеспечения долголетия проекта и роста сообщества. Свобода в данном случае понимается не только в смысле возможности бесплатной публикации программного кода, но и в смысле защиты от доминирующих влияний третьих сторон.

Внешние участники – компании или правительственные группы – регулярно посещают конференции. Проект заинтересован в их данных, в их опыте и вкладе. Однако мы не должны позволять этим заинтересованным сторонам вложить в проект достаточно ресурсов, чтобы иметь возможность влиять на исход голосования в сообществе. На первый взгляд такая предосторожность кажется проявлением паранойи, но в действительности это уже происходило с другими проектами, и сообществу KDE об этом известно. Таким образом, поддержание активности и жизнеспособности проекта свободного ПО напрямую связано с защитой свободы самого проекта. Главная техническая цель – поддержка разработчиков и других участников денежными средства-

ми, материалами, организационными и другими ресурсами – достигается относительно легко.

В результате формируется живое, активное, энергичное сообщество с установившимся процессом разработки, здоровыми изменениями в составе участников и огромным энтузиазмом в работе. В свою очередь, это помогает привлечь и удержать самый важный ресурс в подобных средах: новых участников. KDE удалось переломить типичную тенденцию к созданию все более крупных формальных структур и должностей, и у проекта нет своих диктаторов, какими бы благожелательными они ни были. KDE можно назвать образцовым сообществом свободного ПО.

Чтобы понять, как функционирует это сообщество, как в нем принимаются архитектурные решения и как конкретные процессы влияют на результаты технологического уровня, мы относительно подробно рассмотрим два примера: Akonadi, инфраструктурный уровень управления личной информацией для KDE 4, и ThreadWeaver, небольшую библиотеку для высокоуровневого управления параллельной обработкой.

Akonadi

Платформа KDE 4 – и как платформа разработки, и как среда времени выполнения для запуска интегрированных приложений – покоится на нескольких так называемых «столпах» («pillars»). Эти ключевые инфраструктурные блоки предоставляют основной сервис, который должен быть доступен приложениям для современных рабочих столов. Уровень взаимодействия с оборудованием *Solid* отвечает за передачу рабочему столу информации об устройствах и оповещениях от них (например, появление доступного флэш-диска USB или выход из строя сети). *Phonon* – удобная в программировании мультимедийная прослойка для воспроизведения различных типов информации, а также элементы пользовательского интерфейса для управления этой прослойкой. *Plasma* – библиотека полнофункциональных динамичных масштабируемых (на базе SVG) пользовательских интерфейсов, выходящих за рамки стандартного оформления офисных приложений.

Личная информация пользователя – электронная почта, календарь встреч, задачи, записи в журнале, сообщения в блогах и публикация новостей, закладки, записи чатов, элементы адресной книги и т. д. – содержит не только большой объем первичной информации (содержимое данных). Она формирует богатый контекст, из которого можно многое узнать о предпочтениях пользователя, его социальных взаимоотношениях и рабочих привычках. Этот контекст можно заложить

в основу взаимодействий многих приложений, повышая их ценность и интерес для пользователя – но для этого потребуется надежный, всеобщий и надежный доступ к соответствующей информации. Инфраструктура *Akonadi* создавалась именно для того, чтобы обеспечить доступ к личной информации пользователя, к сопутствующим метаданным и отношениям между всеми данными, а также к сервисным функциям для работы с ними. Akonadi объединяет информацию из различных источников (серверов электронной почты и серверов коллективного ПО, веб-служб и локальных приложений), кэширует и предоставляет доступ к собранным данным. Таким образом, Akonadi является одним из столпов рабочего стола KDE 4, но, как мы вскоре увидим, стремится выйти за эти рамки.

В нескольких ближайших разделах мы изучим историю этой большой и мощной инфраструктуры, познакомимся с социальными, техническими и организационными трудностями, с которыми разработчики сталкивались при ее создании, а также представлениями авторов о ее будущем. Попутно будет представлена дополнительная информация о технических решениях и о причинах, по которым они были выбраны.

История

Еще в самых ранних обсуждениях исходной группы разработчиков KDE по поводу того, какие приложения должны присутствовать в полноценной среде рабочего стола, всегда упоминалась работа с электронной почтой, календарь, управление списками задач и адресная книга. Все соглашались с тем, что это очевидные и важные потребности пользователей. Соответственно KMail, KAddressbook и KOrganizer (приложения для работы с электронной почтой, контактами и событиями/задачами соответственно) вошли в число первых проектов KDE, и первые работоспособные версии появились очень быстро. Потребности пользователей тогда были относительно простыми, а электронная почта принималась либо из локальных накопителей, либо с серверов POP-3. Объем почты был невелик, а почтовые папки обычно хранились в формате mbox (один простой текстовый файл на папку). Сообщество пользователей, на которое был ориентирован проект KDE, неодобрительно относилось к коду HTML в сообщениях, мультимедийный контент любого рода встречался крайне редко, а шифрование и цифровые подписи считались чем-то столь же экзотическим. Нестандартные форматы адресов и календарных данных определялись в текстовом виде, а общий объем хранимой информации вполне поддавался управлению. В то время можно было относительно легко написать базовые приложения, достаточно мощные, чтобы быть принятыми другими

разработчиками KDE, а после первых версий KDE – и сообществом пользователей.

Ранний и продолжительный успех приложений PIM (Personal Information Management), как выяснилось позже, имел и оборотную сторону. Со стремительным развитием Интернета и применения компьютеров вообще пространство задач PIM значительно усложнилось. В программы пришлось интегрировать новые виды доступа к электронной почте (например, IMAP) и форматы хранения (такие как *maildir*). Рабочие группы начали вести совместные календари и адресные книги через серверы коллективного ПО или хранить их локально в новых стандартных форматах вроде vcal/ical или vcard. Каталоги компаний и университетов на серверах LDAP расширились до десятков тысяч записей. Однако пользователи хотели применять уже понравившиеся им приложения KDE в изменившейся ситуации и быстро получить доступ к новым возможностям. В результате (а также из-за того, что активной разработкой приложений PIM занималось небольшое количество людей) архитектурную основу переработать не удалось; ее приходилось постоянно чистить и обновлять с добавлением новых возможностей, а общая сложность кода увеличилась. Основные предположения – что доступ к уровню хранения электронной почты должен быть синхронным и никогда параллельным, что всю адресную книгу можно загрузить в память, что пользователю не приходится часто перемещаться между часовыми поясами – приходилось сохранять и обходить время от времени, потому что затраты на их изменение были неприемлемыми из-за ограничений по времени и ресурсам. Это особенно справедливо для почтового приложения KMail, кодовая база которого постепенно развилась в нечто неприятное, малопонятное, неудобное в расширении и сопровождении и громоздкое. Вдобавок это была стилистически неоднородная подборка работ целой серии авторов, никто из которых не решался вносить слишком глубокие внутренние изменения, опасаясь навлечь на себя гнев своих коллег-разработчиков, если это помешает им читать и писать электронные письма.

Пока приложения PIM находили все более широкое и разнообразное применение, работа над ними велась скорее из чувства долга и верности. Для новых участников эти приложения не обладали ни малейшей привлекательностью. Проще говоря, в KDE и проектах свободного ПО было много других мест, в которых новые участники могли взяться за дело с меньшими начальными усилиями и с меньшим риском – скажем, из-за случайного нарушения какой-нибудь корпоративной функции шифрования, которую они при всем желании не смогли бы протестировать самостоятельно, даже если бы и знали о ее существовании.

Помимо (а может быть, отчасти из-за) технической непривлекательности социальная атмосфера (особенно вокруг KMail) тоже выглядела безрадостно. Дискуссии часто велись грубо, личные выпады встречались сплошь и рядом, а тот, кто пытался присоединиться к группе, редко чувствовал дружественное отношение. Работа над разными приложениями велась в основном в изоляции. В случае KMail даже был довольно неприятный спор из-за права сопровождения. Все это было крайне необычно для сообщества KDE, и внутренняя репутация группы была невысокой.

Однако потребность в интеграции отдельных приложений росла, а пользователи хотели работать с электронной почтой и календарными компонентами из общей оболочки. Участникам, работавшим над этими компонентами, пришлось вносить изменения. Они начали больше общаться, согласовывать интерфейсы, обмениваться планами и графиками и думать о таких аспектах, как продвижение своих разработок в составе пакета *Kontact* и выбор последовательных схем построения имен в своих зонах ответственности. В то же время, из-за коммерческого интереса к приложениям KDEPIM и Kontact внешние заинтересованные стороны требовали более целостного подхода – чтобы сообщество KDEPIM выступало с единых позиций для более профессионального и надежного взаимодействия. Такое развитие ситуации привело к тому, что группа PIM постепенно стала превращаться в одну из самых сплоченных, дружественных и эффективных команд KDE. Их регулярные встречи были образцом для других групп, а многие участники стали друзьями и коллегами. Личные разногласия и прошлые ссоры были забыты, а отношение к новичкам радикально изменилось. Сегодня большая часть общения на темы разработки происходит в объединенном списке рассылки (*kde-pim@kde.org*), разработчики используют общий канал IRC (*#kontact*), а на вопрос, над чем они сейчас работают, отвечают «KDEPIM», а не «KMail» или «KAddressBook».

Управление личной информацией особенно критично в корпоративных условиях. Широкое распространение приложений KDE в крупных организациях породило устойчивый спрос на профессиональные услуги, исправление кода, расширение, упаковку (в дистрибутивах и для конкретных частных сценариев) и т. п.; это, в свою очередь, породило целую экосистему компаний, предлагающих такие услуги. Естественно, компании нанимали людей, наиболее квалифицированных для такой работы – а именно, разработчиков KDEPIM. По этой причине большинство активных участников KDEPIM сейчас трудится над проектом в контексте своей повседневной работы. А среди тех, кому не платят напрямую за работу над KDE, многие являются штатными раз-

работчиками C++ и Qt. Есть и добровольцы, особенно среди новых участников, но основная группа состоит из профессионалов[1].

Из-за несомненной важности инфраструктуры PIM для большинства пользователей (как в личном, так и деловом контексте) руководящим принципом процесса технического принятия решений в KDEPIM стала практичность. Если идею невозможно заставить надежно работать за разумное время, от нее отказываются. Изменения всегда анализируются на предмет потенциального влияния на основную функциональность, которая должна постоянно поддерживаться в рабочем состоянии. Возможности для экспериментов или рискованных решений чрезвычайно малы. В этом отношении проект сильно напоминает группы многих коммерческих и закрытых продуктов и отличается от других частей KDE и проектов свободного ПО вообще. Как упоминалось во Введении, такой подход не всегда позитивен, поскольку он склонен подавлять нововведения и творческий подход.

Эволюция Akonadi

Сообщество KDEPIM регулярно проводит личные встречи, на конференциях и других собраниях разработчиков, а также небольшие «посиделки», на которых группа из 5–10 человек собирается для решения конкретной проблемы за несколько дней интенсивного обсуждения и программирования. Встречи предоставляют отличную возможность для личного обсуждения любых серьезных проблем, принятия важных решений, согласования планов действий и приоритетов. Именно на таких встречах впервые вырабатывается, а затем и укрепляется архитектура больших объединенных проектов, таких как Akonadi. В оставшейся части этого раздела описаны некоторые важные решения, относящиеся к проекту Akonadi, начиная со встречи, на которой были впервые сформулированы его фундаментальные идеи.

Когда группа собралась на своей традиционной зимней встрече в январе 2005 года, в некоторых частях базовой инфраструктуры KDEPIM уже проявлялись признаки перенапряжения. Абстракция, использо-

[1] Кого-то этот факт удивит, поскольку он противоречит интуитивным представлениям о том, что свободное ПО создается в основном студентами с излишками свободного времени. Однако в последнее время все чаще слышатся обоснованные заявления о том, что сейчас созданием и сопровождением большинства успешных проектов свободного ПО занимаются профессиональные программисты. Например, см. Карим Лахани (Karim Lakhani), Боб Вольф (Bob Wolf), Джефф Бейтс (Jeff Bates) и Крис ДиБона (Chris DiBona) «Hacker Survey v0.73» по адресу *http://freesoftware.mit.edu/papers/lakhaniwolf.pdf* (24.6.2002, Boston Consulting Group).

ванная для поддержки внутренних реализаций контактной и календарной информации, KResources, а также уровень хранения данных почтового приложения KMail строились на нескольких базовых предположениях, которые постепенно теряли актуальность. А именно, предполагалось, что:

- Приложений, которым потребуется загружать адресную книгу или календарь, будет очень немного: всего два основных приложения, предназначенных специально для этой цели, KAddressbook и KOrganizer. Также предполагалось, что к хранилищу сообщений электронной почты будет обращаться только KMail. Соответственно поддержка оповещений об изменениях или параллельного доступа либо отсутствовала вовсе, либо была крайне ограниченной, также не было нормального механизма блокировки.
- Объем данных крайне ограничен. В конце концов, со сколькими контактами обычно работает рядовой пользователь, сколько у него встреч или задач? Предполагалось, что порядка «нескольких сотен».
- Доступ к данным потребуется только C++, библиотекам Qt и приложениям KDE.
- Различные внутренние реализации будут работать «в сети», обращаясь к данным, хранящимся на сервере, без их локального дублирования.
- Доступ к данным для чтения и записи будет происходить синхронно и достаточно быстро для того, чтобы пользовательский интерфейс при вызове не блокировался на сколько-нибудь заметное время.

Участники собрания 2005 года согласились с тем, что требования, обусловленные реальными сценариями применения текущей пользовательской базы (и в еще большей степени – возможными требованиями будущих сценариев), не могли быть удовлетворены текущей архитектурой трех основных подсистем. Было очевидно, что в условиях постоянно растущих объемов данных, необходимости организации параллельного доступа для нескольких клиентов, более сложных сценариев обработки ошибок возникала острая нужда в надежных, устойчивых, транзакционных уровнях хранения данных, четко отделенных от пользовательского интерфейса. Также была отмечена желательность использования библиотек KDEPIM на мобильных устройствах, в условиях передачи данных по каналам с низкой пропускной способностью, высокой задержкой и малой надежностью и возможности обращения к данным пользователей не только из приложений, традиционно специализирующихся на работе с такими данными, но и из других мест

рабочего стола. Из этого следовала необходимость предоставления доступа через другие механизмы, отличные от C++ и Qt: сценарные языки, межпроцессные взаимодействия, технологии веб-служб и служб распределенных вычислений[1].

В целом никто не оспаривал все эти высокоуровневые проблемы и задачи. Однако проблемы отдельных команд были гораздо более конкретными и воспринимались по-разному, что привело к разногласиям относительно того, как следует подходить к решению ближайших задач. Например, одна из проблем заключалась в том, что для выборки информации о контакте вся адресная книга должна была загружаться в память. Если адресная книга была большой, содержала много фотографий и других вложений, этот процесс был слишком медленным и расходовал много памяти. Так как доступ к данным осуществлялся через библиотеку, с одним синглетным экземпляром адресной книги на инициализацию библиотеки (а следовательно, на процесс), обычный рабочий стол KDE с почтовым клиентом, адресной книгой и календарем (а также вспомогательными приложениями вроде демона, напоминающего о встречах) легко мог загрузить адресную книгу в память в четырех и более экземплярах.

Для решения непосредственной проблемы дублирования экземпляров адресной книги в памяти была предложена схема на базе «клиент/сервер». В двух словах, данные будут храниться в памяти только одним процессом. После однократной загрузки с диска весь доступ к данным будет осуществляться с использованием механизмов IPC – прежде всего *DCOP*, тогдашней инфраструктуры вызова удаленных процедур в KDE. Кроме того, при таком решении все взаимодействия с внутренними подсистемами контактной информации (например, серверами коллективного ПО) изолировались в одном месте – так решалась еще одна проблема старой архитектуры. Однако в ходе длительного обсуждения выяснилось, что, несмотря на решение проблемы с затратами памяти, остается несколько других серьезных проблем. Прежде всего, механизмы блокировки, разрешения конфликтов и оповещения об изменениях необходимо было как-то реализовать поверх сервера. Также было высказано мнение, что полиморфные (а, следовательно, основанные на указателях) API, особенно интенсивно используемые в календаре, слишком заметно усложнят сериализацию, необходимую для передачи данных средствами DCOP. Постоянная передача данных через интерфейс IPC могла оказаться слишком медленной, и это тоже

[1] *http://gdp.globus.org/gt3-tutorial/multiplehtml/ch01s03.html – Примеч. перев.*

вызвало беспокойство, особенно если доступ к почте должен был осуществляться через сервер, как предлагалось в одном из вариантов.

Собрание пришло к выводу, что проблему с затратами памяти лучше решать за счет более умного механизма как совместного использования дискового кэша, так и представления в памяти – возможно, с использованием файлов, отображаемых на память (memory mapped files). Сложность перехода на архитектуру «клиент/сервер» выглядела слишком серьезной, и участники решили, что выгоды от нее не оправдают риск от разрушения работоспособной (пусть и не лучшей) системы. В качестве возможной альтернативы было решено присмотреться к серверу EDS (Evolution Data Server), который использовался конкурирующим пакетом PIM из проекта GNOME и был написан на C с использованием *glib* и библиотек GTK.

Сторонники идеи сервера данных были несколько разочарованы, и за несколько следующих месяцев особого движения в ту или иную сторону не наблюдалось. Короткий экскурс в кодовую базу EDS с целью включения поддержки библиотек, использовавшихся KDE для работы с адресной книгой, завершился быстро и внезапно. Исследователи поняли, что решение, основанное на объединении миров C и C++, прежде всего их менталитетов и стилей API, будет в лучшем случае неэлегантным, а в худшем – неполным. Кроме того, в EDS применялась технология *CORBA*, изначально принятая KDE, а потом по разным причинам замененная на DCOP; это тоже не добавляло энтузиазма. Следует честно признать, что отказ от EDS как основы для новой инфраструктуры данных KDE был обусловлен как техническим анализом, так и личными предубеждениями против C, неприязнью к реализации и предполагаемыми проблемами с сопровождением, наряду с определенной долей синдрома «изобретено не у нас».

К концу 2005 года проблемы, обсуждавшиеся в начале года, стали еще более насущными. Электронная почта стала труднодоступной для таких приложений, как поисковые агенты или инфраструктуры семантической пометки и связи. Чтобы получить доступ к вложению в сообщении, найденном по поисковому индексу, приходилось запускать почтовое приложение вместе с пользовательским интерфейсом и открывать сообщение для редактирования. Эти и другие аналогичные проблемы с удобством пользования и эффективностью работы только повышали недовольство существующей инфраструктурой.

На конференции в Бангалоре (Индия), где в то время находилась значительная часть группы разработчиков Evolution, мы обсудили с ними некоторые существующие проблемы и расспросили об опыте использования EDS. На этих встречах стало ясно, что они столкнулись со мно-

гими проблемами, выявленными группой KDEPIM, рассматривали аналогичные решения и пришли к выводу, что расширение EDS для поддержки почты или портирование с уходом от CORBA не были приемлемыми вариантами. В целом группа Evolution высказалась примерно так: если разработчики KDEPIM построят новую инфраструктуру для доступа к данным PIM, то они будут заинтересованы в совместном использовании ее, по крайней мере, на концептуальном уровне, а то и на уровне реализации (при условии, что она не будет реализована исключительно на C++ как библиотека KDE).

Возможность будущего совместного использования столь важного инфраструктурного блока всеми рабочими столами в сфере свободного ПО, а также факт, что при правильной реализации такой проект окажется полезным более широкому сообществу разработчиков за пределами KDEPIM, придал новый вес идее архитектуры «клиент/сервер». Такая архитектура предоставит точку интеграции в виде протокола или механизма IPC (вместо библиотеки, с которой нужно компоновать код), а это откроет путь для других инструментариев, языков и парадигм. Сценарий сервера PIM, общего для KDE и GNOME, начал выглядеть все более реалистично в свете появления стека *DBUS* – кроссплатформенной системы IPC, интегрируемых с рабочими столами. На тот момент технология DBUS активно интегрировалась в оба проекта сред рабочих столов, а ее координация осуществлялась сайтом *freedesktop.org*.

Еще один любопытный пример «межпроектного опыления»: в том году на конференции присутствовали разработчики из проекта баз данных PostgreSQL, участвовавшие в некоторых дискуссиях по теме. Они заинтересовались нашим проектом с точки зрения эффективного управления, организации доступа и возможности обращения с запросами к большому объему структурированных данных (электронная почта, события и контакты пользователя). Было решено, что их опыт и создаваемый программный продукт могли бы стать важной частью такой системы. Они привлекли наше внимание ко многим интересным аспектам, относящимся к эффективной организации поиска и проектированию системы с возможностью расширения типов, но также упомянули и более «приземленные» проблемы (скажем, как организовать архивацию в такой системе и как обеспечить надежность и целостность данных).

Через несколько месяцев концепция сервера данных PIM снова была представлена публике на ежегодной встрече в Оснабрюке (Германия) – на этот раз с дополнительной поддержкой группы, работающей над электронной почтой. Эта группа скептически отнеслась к идее год назад и проявляла наибольшую обеспокоенность по поводу возможного

снижения производительности и повышения сложности[1]. Новые перспективы за пределами проекта KDE; тот факт, что альтернативные решения, предложенные год назад, не были реализованы (и даже проверены); постоянно возрастающее давление на группу с требованием разобраться со своими проблемами – все это заставило противников концепции пересмотреть свою позицию и серьезно отнестись к этому рискованному изменению.

Немалую помощь оказал и нетехнический аспект, который сформировался во многих беседах с разработчиками за прошедший год. Группа осознала, что их главной проблемой была нехватка новых разработчиков, приходящих в проект, а это обстоятельство в значительной мере объяснялось громоздкими и неудобными библиотеками и приложениями. Существовали опасения, что текущие разработчики не смогут участвовать в проекте вечно, что они не смогут передать все свои знания следующему поколению для поддержания жизни проекта. Было решено направить как можно больше внимания и энергии на то, чтобы воплотить в программном коде объединенный опыт и знания всех, кто работал над KDEPIM. Созданный продукт должен был стать основой для следующего поколения участников, а другие разработчики из проекта KDE и за его пределами должны были строить на его основе улучшенные программы PIM. Конечной целью было создание программ, над которыми было бы интересно работать; программ, лучше документированных, более понятных и более современных. Мы надеялись, что это привлечет новых участников и позволит снова организовать творческую работу в пространстве PIM, оградив от многих неприятностей и сложностей тех, кто захочет воспользоваться инфраструктурной формой. Похоже, архитектура «клиент/сервер» способствовала решению этой задачи.

Следует отметить, что фундаментальное решение о разрушающей переработке всей инфраструктуры хранения данных для KDEPIM фактически означало необходимость заново переписать большие фрагменты системы. Мы вполне сознательно согласились с тем, что у нас останется значительно меньше ресурсов для сопровождения текущей кодовой базы, ее поддержания в стабильном и рабочем состоянии, а также ее включения в состав KDE 4.0, поскольку это событие почти наверняка должно было произойти до завершения любой сколько-нибудь серьезной переработки.

Как выяснилось позже, это привело к необходимости выпуска KDE 4.0 без KDEPIM – неприятная, но, скорее всего, необходимая жертва.

---

[1] *http://pim.kde.org/development/meetings/osnabrueck4/overview.php*

Когда группа согласилась с общим направлением работы, была опубликована следующая формулировка задачи:

> «Мы намерены спроектировать расширяемую универсальную службу хранения данных PIM и метаданных с поддержкой параллельного доступа для чтения, записи и выборки данных. Она должна обеспечивать однозначную идентификацию и выборку объектов в масштабах среды рабочего стола».

Архитектура Akonadi

Некоторые ключевые аспекты, оставшиеся в более поздних итерациях архитектуры, уже присутствовали в первом плане архитектуры, который был разработан на собрании. Главным среди них было решение отказаться от использования DBUS, очевидного выбора механизма межпроцессных взаимодействий в Akonadi, в качестве транспортного механизма полезных данных. Вместо этого для выполнения групповой пересылки использовался отдельный транспортный канал и протокол, а именно IMAP. Одной из причин в пользу такого решения была возможность внеполосного управления трафиком; например, это позволяло отменять слишком затянувшуюся пересылку данных, так как управляющий канал никогда не блокируется каналом данных. В IMAP пересылка данных требует гораздо меньших затрат по сравнению с пересылкой через механизм IPC, так как протокол проектировался для быстрой потоковой пересылки больших объемов данных. Решение учитывало специфические характеристики производительности DSBUS, поскольку в документации DBUS было явно указано, что этот механизм не предназначен для подобных сценариев использования. Мы получали возможность использования существующего библиотечного кода IMAP, избавляя от хлопот по реализации протокола как саму команду KDEPIM, так и будущих сторонних разработчиков, желающих интегрироваться с Akonadi. Также сохранялась возможность обращения к содержимому почтового хранилища из обобщенных, не связанных с Akonadi инструментариев – например, почтовых приложений командной строки (таких как *pine* или *mutt*). Это помогало преодолеть субъективные страхи пользователей, опасающихся доверять свои данные системе, блокирующей доступ к информации любыми другими средствами. Так как IMAP поддерживает только работу с электронной почтой, протокол требовалось расширить для поддержки других типов MIME, но мы сочли эту задачу выполнимой без потери совместимости с базовым протоколом. Также обсуждалась возможность реализации транспорта на базе http/webdav с возможным применением существующих реализаций серверов http (например, Apache), но такой подход не получил поддержки.

Важнейшее свойство Akonadi заключается в том, что система образует единый центральный кэш для всех данных PIM и связанных с ними метаданных. Если в старой инфраструктуре типичным сценарием считался сетевой доступ к внутренним хранилищам, Akonadi ориентируется на локальную копию. Последняя создается в тот момент, когда данные потребуется вывести для пользователя (например), и в ней сохраняется максимально возможный объем уже загруженной информации для предотвращения загрузки лишних данных. Предполагается, что приложения загружают в память только те данные, которые действительно необходимы для вывода (в случае почтового клиента – заголовки нескольких сообщений, видимых в данный момент в текущей папке) и не пытаются самостоятельно организовать кэширование. Такой подход открывает возможность таких оптимизаций, как совместное использование кэшей, обеспечение их логической целостности, сокращение затрат памяти приложений и отложенная загрузка данных, которые не требуется выводить немедленно. Так как кэш присутствует всегда, хотя и может быть незаполненным, появляется возможность оффлайнового использования (по крайней мере, во многих случаях). Это в значительной степени повышает устойчивость приложения перед проблемами, обусловленными ненадежными, медленными каналами с высокой задержкой.

Для обеспечения неблокирующего параллельного доступа сервер проектировался таким образом, чтобы с каждым подключением ассоциировался контекст выполнения (программный поток), а на всех уровнях архитектуры учитывается потенциально большое количество параллельных контекстов. Отсюда вытекает транзакционная семантика операций с состоянием, возможность выявления сочетаний операций, приводящих к нарушению целостности состояния, и многое другое. Кроме того, данное решение определяет ключевое ограничение на выбор технологии управления хранением данных системы на диске – а именно поддержку интенсивного параллелизма в операциях как чтения, так и записи. Поскольку состояние может быть изменено в любой момент параллельными сеансами (подключениями), механизм оповещения о таких изменениях должен быть надежным, полнофункциональным и быстрым. Это еще одна причина для отделения управляющей информации, для которой характерна низкая задержка, низкая пропускная способность канала, но высокая критичность, от массовой пересылки данных с более высокой задержкой, более высокой пропускной способностью канала, но меньшей критичностью – при таком разделении оповещения не будут «застревать» в то время, пока приложение загружает большое вложение в сообщении электронной почты. Данная способность становится еще более актуальной, если приложе-

ние способно обрабатывать внеполосные оповещения одновременно с пересылкой данных. Возможно, большинство приложений для потенциальных пользователей Akonadi, существующих в настоящее время, такой возможностью не обладает, но можно с полным основанием предположить, что в будущих приложениях роль параллельной обработки только возрастет – не в последнюю очередь благодаря появлению таких инструментов, как ThreadWeaver (см. следующий раздел). Высокоуровневые вспомогательные классы, входящие в библиотеку доступа к Akonadi из KDE, уже используют эту возможность.

Уже в первой итерации архитектуры присутствовал и другой фундаментальный аспект Akonadi: компоненты, обеспечивающие доступ к определенной разновидности внутренних хранилищ (скажем, к серверу коллективного ПО), выполняются как отдельные процессы. Такое решение обладает рядом преимуществ. Потенциально ненадежные и медленные коммуникации с сервером не могут поставить под угрозу стабильность всей системы. Сбой агента[1] не приводит к сбою всего сервера. Если синхронное взаимодействие с сервером более удобно (а, может быть, является единственно возможным способом получения данных), процесс блокируется без блокировки любых других взаимодействий с Akonadi. Код агентов может компоноваться со сторонними библиотеками без привязки к базовой системе и может лицензироваться отдельно; это важно для тех ситуаций, когда библиотеки доступа не существуют в свободном ПО. Агенты изолируются от адресного пространства сервера Akonadi, а, следовательно, создают меньше потенциальных угроз для безопасности. Такой подход упрощает их разработку третьими сторонами, их развертывание и тестирование в рабочих системах с применением любого подходящего языка программирования (при наличии поддержки DBUS и IMAP). Конечно, есть и оборотная сторона: необходимость сериализации данных на пути к хранилищу Akonadi, поскольку этот путь пересекает границы процессов. Впрочем, на практике эта проблема не столь значительна по двум причинам. Во-первых, с точки зрения пользователя все происходит в фоновом режиме, без прерывания каких-либо операций на уровне пользовательского интерфейса. Во-вторых, данные либо уже доступны локально в кэше, когда пользователь запросит их, либо поступят из сетевого сокета, который может передавать данные сокету сервера Akonadi – во многих случаях в неразобранном виде, а теоретически

[1] Сущности, взаимодействующие с сервером Akonadi для чтения и записи данных, называются агентами (стандартный пример – агент сбора данных). Агенты, специализирующиеся на синхронизации данных между локальным кэшем и удаленным сервером, называются ресурсами.

даже без копирования. Запрос данных пользователем для вывода – одна из тех операций, которые должны выполняться как можно скорее, чтобы избежать заметного времени ожидания. Во многих случаях взаимодействия между агентами и хранилищем не критичны по быстродействию.

С точки зрения параллелизма Akonadi состоит из двух уровней. На уровне мультиобработки каждый пользователь Akonadi (обычно приложение или агент) обладает отдельным адресным пространством и отдельным контекстом захвата ресурсов (файлы, сетевые сокеты и т. д.). Каждый из этих процессов может открыть одно или несколько подключений к серверу и каждый представлен во внутренней реализации отдельным программным потоком. Чтобы избежать неоднозначного выбора между потоками и процессами, мы используем и то, и другое: процессы используются там, где важна надежность, ресурсы и изоляция безопасности, а потоки – там, где по соображениям производительности необходимо общее адресное пространство, и где код находится под управлением реализации самого сервера Akonadi (а, следовательно, где проблемы стабильности менее вероятны).

Одной из первых желательных характеристик системы, определенных при обсуждении, была возможность добавления поддержки новых типов данных с относительно небольшими усилиями. В идеале уровень управления данными должен быть полностью независим от типа, а информация о содержимом и оформлении данных должна быть централизована для каждого типа данных (сначала электронная почта, события и контакты, позднее заметки, RSS-потоки, записи IM, закладки, а, возможно, и многое другое). С самого начала было понятно, что мы должны стремиться к этой цели – неясно было, как ее достичь. Лишь после нескольких итераций архитектуры базовой системы и API библиотеки доступа нам удалось справиться с этой задачей. Конечный результат будет описан чуть позднее в этом разделе.

Во время начальных обсуждений архитектуры группа рассматривала общую картину довольно традиционно, разделяя ее на уровни, соответствующие сферам влияния. Как показывает копия рисунка из обсуждения на рис. 12.1, внизу располагается уровень хранения данных, уровнем выше – логика доступа к данным, еще выше – уровень транспорта и протокола доступа, и так далее до пространства приложений.

Несмотря на ведущиеся дискуссии по поводу того, как должен выглядеть API для обращения приложений к хранилищу (в отличие от API, используемого агентами или ресурсами в системе), некоторые участники с подозрением относились к самому факту существования единого API доступа, используемого всеми сущностями, взаимодействую-

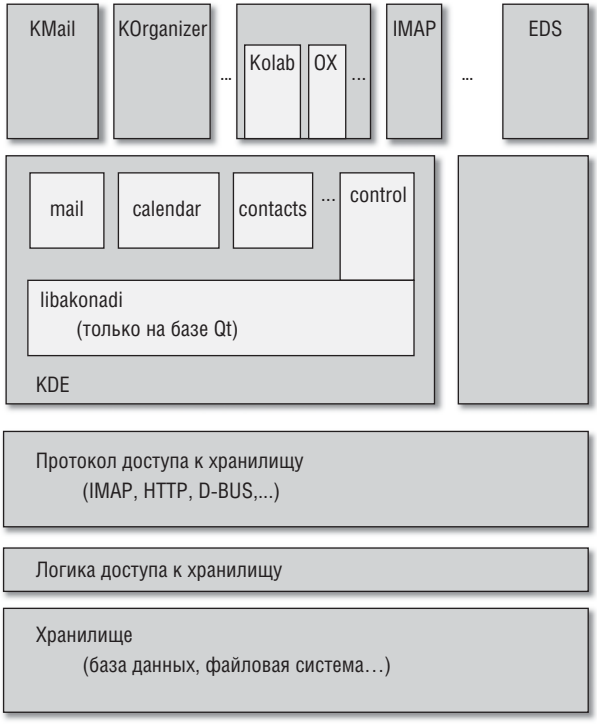


Рис. 12.1. *Исходная схема уровней в архитектуре Akonadi*

щими с хранилищем, – предназначенными как для предоставления данных, так и для работы с данными с точки зрения пользователя. Быстро выяснилось, что этот вариант действительно является предпочтительным, поскольку он упрощает архитектуру и делает ее более простой и симметричной. Любая операция агента с данными инициирует оповещения об изменениях, которые воспринимаются всеми остальными агентами, отслеживающими эту часть системы. Любые потребности ресурсов в дополнение к потребностям приложений, например способность поставлять в хранилище большие объемы данных без генерирования каскада оповещений, должны быть достаточно универсальными и практичными для включения в единый унифицированный API. Но при этом необходимо каким-то образом сохранить достаточную простоту API как для доступа приложений, так и для ресурсов, чтобы сторонние разработчики могли предоставить дополнительные внутренние подсистемы для серверов коллективного ПО, а разработчики прило-

жений приняли архитектуру Akonadi. Все необходимые особые случаи, относящиеся к быстродействию или восстановлению после ошибок, должны по возможности обрабатываться незаметно для пользователя. Проблема предотвращения каскадов оповещений решается настраиваемой системой сжатия оповещений и отслеживания обновлений, на которую пользователи системы могут подписаться с выбранным ими уровнем детализации.

Эта концепция представлена на следующей версии высокоуровневой архитектурной диаграммы, показанной на рис. 12.2; на ней уровни системы изображены в виде концентрических колец или их частей.

Из описанных выше требований достаточно очевидно следует, что реляционная база данных значительно упростит реализацию нижнего (или внутреннего) уровня архитектуры. По крайней мере, для метаданных, связанных с фактическими элементами PIM (время выборки, локальные теги, политики уровня папок и т. д.) – типизированных, структурированных, очевидным образом выигрывающих от быстрого, индексированного произвольного доступа и эффективных запросов –

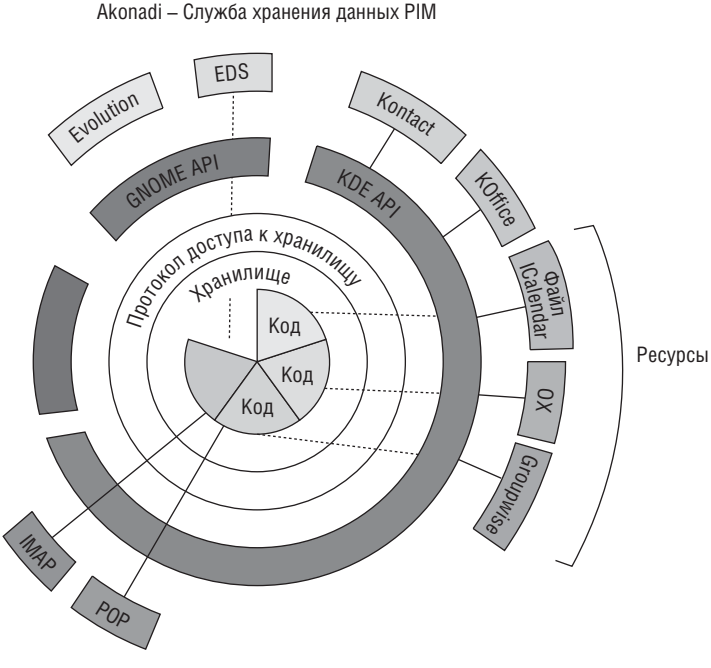


Рис. 12.2. *Кольцевая диаграмма (вместо иерархической)*

другие решения серьезно не рассматривались. Что касается самих полезных данных (сообщений электронной почты, контактов и т. д.) и организации их хранения на диске, решение не столь очевидно. Поскольку хранилище не должно зависеть от типа, таблица базы данных, в которой хранится информация, не сможет делать какие-либо допущения по поводу структуры данных и фактически будет вынуждена хранить их в полях двоичных объектов (BLOB). При работе с неструктурированными (с точки зрения базы) данными теряется часть преимуществ от использования баз данных. Эффективное индексирование становится практически невозможным, так как для него необходим разбор содержимого полей данных. Соответственно запросы к таким полям будут работать плохо. Ожидаемые схемы доступа также неблагоприятны для базы данных; предпочтительным вариантом был бы механизм, обеспечивающий постоянную потоковую передачу данных. Отказ от использования базы данных будет означать, что транзакционную семантику операций с хранилищем придется реализовывать вручную. В качестве решения проблемы было предложено расширение принципов стандарта *maildir*, которое фактически делало возможным неблокируемый доступ ACID за счет использования атомарных операций переименования файловой системы[1]. В первом цикле реализации было решено, что в базе данных будут храниться как данные, так и метаданные, а оптимизация будет проведена позднее, когда требования к поиску определятся более четко.

Так как хранилище Akonadi представляет собой кэш с личной информацией пользователя, оно позволяет реализовать эффективное управление жизненным циклом кэшируемых данных. Концепция политики кэширования предоставляет в распоряжение пользователя очень точные средства контроля над тем, какие элементы и в течение какого времени хранятся в кэше. На одном конце спектра (скажем, для встроенных устройств с плохим каналом связи и очень ограниченной продолжительностью) логично определить политику, которая запрещает хранить что-либо, кроме минимальной заголовочной информации. Полные элементы данных загружаются только тогда, когда возникнет необходимость в их выводе. При этом уже загруженные элементы хранятся в памяти, пока не будет разорвано подключение или не отключится питание памяти. Портативный компьютер, который часто имеет ненадежную связь, но значительный объем дискового пространства, может заранее кэшировать как можно больший объем данных на диске, чтобы сделать возможной работу в оффлайне; далее остается лишь очищать некоторые папки с периодичностью в несколько дней, недель

[1] *http://pim.kde.org/development/meetings/osnabrueck4/icaldir.php*

и даже месяцев. С другой стороны, настольной рабочей станции с постоянно доступным широкополосным подключением к Интернету или серверу коллективного ПО через локальную сеть можно рассчитывать на быстрый сетевой доступ к данным. Если только пользователь не захочет сохранить локальную копию в целях архивации или снижения загрузки канала связи, кэширование может вестись в более пассивном режиме, например с сохранением только загруженных вложений для локального индексирования и обращения. Такие политики кэширования могут устанавливаться на уровне папок, учетных записей или внутренних подсистем, а их соблюдение обеспечивается компонентами, работающими на сервере в отдельном потоке с низким приоритетом. Эти компоненты регулярно просматривают базу данных в поисках данных, которые могут быть очищены в соответствии со всеми относящимися к ними политиками.

Среди важнейших недостающих компонентов, выявленных в архитектуре на встрече в 2007 году, были реализация поиска и семантических связей. На платформе KDE 4 появлялись мощные решения в области индексирования, расширенной обработки метаданных и семантических сетей в проектах Strigi и Nepomuk. Их возможная интеграция с Akonadi открывала очень интересные возможности. Тогда было неясно, удастся ли реализовать компонент, поставляющий данные Strigi для полного индексирования, в виде агента, в виде отдельного процесса, работающего на основании оповещений от ядра, или его придется интегрировать в серверное приложение по соображениям производительности. Так как в этом случае как минимум полный текстовый индекс будет храниться за пределами Akonadi, возникали сопутствующие вопросы – как организовать разбиение поисковых запросов, как и где будут объединяться результаты Strigi и Akonadi, как запросы будут передаваться внутренним серверам с поддержкой сетевого поиска (например, серверам LDAP)? Также было необходимо обсудить разделение обязанностей с реализацией Nepomuk – например, нужно ли полностью делегировать ей работу с тегами? В какой-то степени эти обсуждения продолжаются до сих пор, так как задействованные технологии развиваются вместе с Akonadi, а текущие решения все еще не были проверены в условиях реальной эксплуатации. На момент написания книги существуют агенты, поставляющие данные Nepomuk и Strigi; это отдельные процессы, использующие тот же API к хранилищу, что и остальные клиенты и ресурсы. Входные поисковые запросы формулируются на XESAM или *SPARQL* – языках запросов Strigi и Nepomuk соответственно, которые также реализованы другими поисковыми системами (например, Beagle), и передаются им через механизм DBUS. Передача происходит внутри серверного процесса Akonadi.

Результаты поступают через DBUS в виде списка идентификаторов, которые Akonadi использует для представления результатов как фактических данных из хранилища. Само хранилище на текущий момент не поддерживает никаких средств поиска или индексирования.

Формирование API для библиотеки доступа, написанной на C++ и специфической для KDE, заняло некоторое время – в основном потому, что с самого начала было неясно, насколько независимым от типа будет сервер и какой объем информации типа будет раскрыт в библиотеке. К апрелю 2007 года стало ясно, что расширение библиотеки доступа для поддержки новых типов будет осуществляться при помощи специальных модулей расширения, называемых сериализаторами. Они представляют собой библиотеки, загружаемые на стадии выполнения и способные преобразовывать данные определенного формата, определяемого типом MIME, в двоичное представление для хранения в формате BLOB на сервере, и наоборот – восстанавливать представление в памяти по сериализованным данным. Эти функции ортогональны добавлению поддержки новой внутренней подсистемы хранилища и используемых ей форматов данных, что делается посредством реализации нового ресурсного процесса (агента). Ресурс отвечает за преобразование данных, передаваемых сервером по каналу связи, в типизованное представление в памяти, с которым он умеет работать. Далее сериализатор используется для его преобразования в двоичный поток данных, который может быть помещен в хранилище Akonadi, и его обратного преобразования при выборке библиотекой доступа. Сериализатор также может разделять данные на несколько частей, чтобы обеспечить возможность частичного доступа (скажем, только к телу сообщения или только к вложениям). Центральный класс этой библиотеки `Akonadi::Item` представляет отдельный элемент данных в хранилище. Он обладает уникальным кодом, позволяющим однозначно идентифицировать его на компьютере пользователя, а также ассоциировать с другими сущностями в составе семантических связей (например, с удаленными идентификаторами). Так с элементом данных связывается информация о местонахождении в хранилище, атрибуты, полезные данные и кое-какая полезная инфраструктура (скажем, флаги и счетчик версий). Атрибуты и полезные данные обладают сильной типизацией, а методы для их чтения/записи создаются на основе шаблона. Сами экземпляры `Akonadi::Item` легко копируются с минимальными затратами ресурсов. Элемент данных параметризуется по типу полезных данных и атрибутов, но при этом сам не обязан быть шаблонным классом. Для реализации этой возможности применяются довольно нетривиальные трюки с шаблонами, но итоговый API очень прост в использовании. Чтобы избежать нечеткой семантики принадлежности, пред-

полагается, что полезные данные относятся к типу-значению (value type). В тех случаях, когда полезные данные должны быть полиморфными (а, следовательно, представляют собой указатель), или если для работы с некоторым типом данных уже существует библиотека на базе указателей (пример – *libkcal*, библиотека для работы с событиями и управления задачами в KDE), для обеспечения предоставления семантики значений могут использоваться общие указатели, такие как `boost::shared_ptr`. Попытка использования низкоуровневого указателя в качестве полезных данных выявляется посредством специализации шаблонов и приводит к нарушениям проверочных условий времени выполнения.

Следующий пример показывает, как легко добавить поддержку нового типа в Akonadi, если у вас уже имеется библиотека для работы с данными в этом формате, как это часто бывает. Ниже приведен полный исходный код сериализатора для контактов, или объектов `KABC::Addresse`, как они называются в библиотеке KDE:

```
bool SerializerPluginAddressee::deserialize( Item& item,
                                             const QByteArray& label,
                                             QIODevice& data,
                                             int version )
{
    if ( label != Item::FullPayload || version != 1 )
        return false;

    KABC::Addressee a = m_converter.parseVCard( data.readAll() );
    if ( !a.isEmpty() ) {
        item.setPayload<KABC::Addressee>( a );
    } else {
        kWarning() << "Empty addressee object!";
    }
    return true;
}

void SerializerPluginAddressee::serialize( const Item& item,
                                           const QByteArray& label,
                                           QIODevice& data,
                                           int &version )
{
    if ( label != Item::FullPayload
      || !item.hasPayload<KABC::Addressee>() )
        return;
    const KABC::Addressee a = item.payload<KABC::Addressee>();
    data.write( m_converter.createVCard( a ) );
    version = 1;
}
```

Типизированные полезные данные, методы `setPayload` и `hasPayload` класса `Item` позволяют разработчикам легко и непосредственно использовать «родные» типы своих библиотек типов данных. Взаимодействия с хранилищем обычно представляются в виде *заданий* (*jobs*), применений паттерна «команда». Задания отслеживают жизненный цикл операции, предоставляют точку отмены и доступ к контекстам ошибок, а также позволяют следить за ходом выполнения. Класс `Monitor` позволяет клиенту следить за изменениями в хранилище на нужном уровне (например, типов MIME, коллекций или даже только определенных частей конкретных элементов данных). Следующий пример из миниприложения, оповещающего о поступлении электронной почты, демонстрирует эти концепции. В данном случае полезные данные являются полиморфными и инкапсулируются в общем указателе:

```
Monitor *monitor = new Monitor( this );
monitor->setMimeTypeMonitored( "message/rfc822" );
monitor->itemFetchScope().fetchPayloadPart( MessagePart::Envelope );
connect( monitor, SIGNAL(itemAdded(Akonadi::Item,Akonadi::Collection)),
         SLOT(itemAdded(Akonadi::Item)) );
connect( monitor, SIGNAL(itemChanged(Akonadi::Item,QSet<QByteArray>)),
         SLOT(itemChanged(Akonadi::Item)) );

// Начать исходную загрузку первого отображаемого сообщения
ItemFetchJob *fetch = new ItemFetchJob( Collection( myCollection ), this );
fetch->fetchScope().fetchPayloadPart( MessagePart::Envelope );
connect( fetch, SIGNAL(result(KJob*)), SLOT(fetchDone(KJob*)) );

...

typedef boost::shared_ptr<KMime::Message> MessagePtr;

void MyMessageModel::itemAdded(const Akonadi::Item & item)
{
    if ( !item.hasPayload<MessagePtr>() )
       return;
    MessagePtr msg = item.payload<MessagePtr>();
    doSomethingWith( msg->subject() );
    ...
}
```

Первая версия и дальнейшая история

Когда группа снова собралась в холодном и дождливом Оснабрюке в январе 2008 года, приглашенные разработчики смогли представить первые примеры использования Akonadi в приложениях. Авторы Mailody, конкурента стандартного почтового приложения KDE, заранее решили, что Akonadi поможет им построить более качественное прило-

жение, и первыми опробовали новые возможности и API. Полученная от них обратная связь помогла определить, что все еще оставалось слишком сложным, где нужно было предоставить дополнительную информацию, а какие концепции были недостаточно хорошо документированы или реализованы. На встрече также присутствовал Кевин Краммер (Kevin Krammer), еще один из ранних сторонников Akonadi. Он взялся за интересную задачу: попытаться предоставить пользователям старых библиотек данных PIM из проекта KDE доступ к Akonadi (и наоборот – обеспечить работу с данными, хранимыми в старой инфраструктуре через Akonadi) при помощи агентов совместимости и ресурсов для обеих инфраструктур. Проблемы, с которыми он столкнулся при решении этой задачи, выявили некоторые пробелы в API, а также подтвердили, что новый инструментарий будет, как минимум, поддерживать всю существующую функциональность.

Примечательным результатом этой встречи стало решение об отказе от обратной совместимости с IMAP на уровне протокола. Проект так далеко ушел от исходного стандарта, рассчитанного только на работу с электронной почтой, что сохранение способности сервера Akonadi работать в режиме стандартного сервера IMAP для обеспечения доступа к электронной почте превратилось в балласт, который перевешивал преимущества этой функции. Протокол IMAP послужил хорошей отправной точкой, и многие из его концепций остались в протоколе доступа Akonadi, но применительно к последнему название IMAP стало уже необоснованным. Возможно, этот механизм вернется в будущих версиях сервера, например в виде реализации прокси-сервера совместимости.

Сроки выхода KDE 4.1 быстро приближались. Группа снова встретилась в марте 2008 года для проведения полного анализа API перед его первым выпуском, после которого им придется поддерживать его стабильность и двоичную совместимость в обозримом будущем. За два дня было выявлено огромное количество мелких и крупных несоответствий, отсутствующих фрагментов документации, странностей реализации и неудачно выбранных имен. Все эти ошибки были исправлены за несколько следующих недель.

На момент написания книги подходит время выпуска версии KDE 4.1, а команда Akonadi с волнением ожидает реакции многих разработчиков приложений и библиотек из сообщества KDE, составляющих нашу целевую аудиторию. Интерес к написанию ресурсов доступа для различных видов хранилищ растет, а энтузиасты уже начали работу над поддержкой адресных книг Facebook, закладок Delicious, серверов MS Exchange и коллективного ПО через библиотеку OpenChange, блоговых поставок RSS и других источников. Интересно посмотреть, что

сможет создать сообщество, когда доступ к данным из этих и многих других источников станет простым, повсеместным и надежным; когда появится эффективная поддержка запросов; когда данные можно будет снабжать аннотациями, связывать элементы данных друг с другом, создавать смысловое содержание и контекст; когда разработчики смогут использовать все богатство возможностей, чтобы пользователи извлекали больше пользы из своих программ и получали больше удовольствия от работы с ними.

Две взаимосвязанные идеи оптимизации до сих пор не были реализованы. Первая – отказ от хранения полезных данных в двоичных объектах базы данных за счет хранения в таблице только URL-адреса файловой системы, тогда как сами данные будут храниться непосредственно в файловой системе, как упоминалось выше. Развивая эту тему, можно избежать копирования данных из файловой системы в память, передавая их через сокет для доставки клиенту (другой процесс), тем самым создавая вторую копию в памяти только для освобождения первой копии. Задача может решаться передачей приложению файлового дескриптора (handle), чтобы приложение могло отобразить файл в память для обращения. Пока трудно сказать, удастся ли это сделать без нарушения надежности, целостности, безопасности и ограничений API, обеспечивающих работу архитектуры. Альтернативный вариант – воспользоваться расширением MySQL для потоковой передачи двоичных объектов, обещающим сохранить большую часть преимуществ от использования API реляционной базы данных, совместив их с большей частью преимуществ низкоуровневого доступа к файловой системе.

Хотя сервер и клиентские библиотеки KDE будут впервые опубликованы в KDE 4.1, группа по-прежнему стремится как можно шире поделиться своими разработками с миром свободного ПО. Для этого на сайте *freedesktop.org* был создан специальный проект, и сервер будет перемещен туда сразу же после завершения процесса. Всем интерфейсам DBUS были присвоены имена, нейтральные по отношению к среде рабочего стола; единственной зависимостью от сервера является библиотека Qt в версии 4, которая является частью стандартной базовой спецификации Linux и распространяется по лицензии GNUGPL в версиях для Linux, Windows, OS X и встроенных систем, включая Windows CE. Следующим серьезным шагом будет реализация второй библиотеки доступа – например, на языке Python, хорошая инфраструктура которого позволит решить эту задачу с относительно небольшими усилиями; а, может быть, на языке Java, обладающем тем же свойством.

ThreadWeaver

ThreadWeaver сейчас входит в число базовых библиотек KDE 4. Она рассматривается здесь потому, что ее происхождение во многих отношениях отличается от происхождения проекта Akonadi, и их интересно сравнить. Библиотека ThreadWeaver занимается планированием параллельных операций. Она была задумана в те времена, когда ее функции было технически невозможно реализовать на базе библиотек, использовавшихся в KDE, а именно Qt. Необходимость в ней сознавали многие разработчики, но лишь после выхода Qt 4 библиотека «дозрела» и получила широкое распространение. Сейчас она используется в таких серьезных приложениях, как KOffice и KDevelop. Обычно она применяется в крупномасштабных, более сложных программных системах, в которых необходимость параллельной и внеполосной обработки становится особенно насущной.

ThreadWeaver – планировщик параллельно выполняемых заданий. Целью библиотеки является управление и распределение ресурсов в многопоточных программных системах. Вторая цель заключается в том, чтобы снабдить прикладных программистов инструментами для реализации параллелизма, сходными со средствами разработки приложений GUI. Впрочем, эти цели относятся к высокому уровню; существуют и другие, менее масштабные: предотвращение синхронизации методом «грубой силы» и предоставление средств для кооперативной сериализации доступа к данным; использование возможностей современных библиотек C++ (таких как потоковая безопасность, неявный совместный доступ и коммуникации «сигнал/слот»); интеграция с графическим интерфейсом приложения за счет отделения обрабатывающих элементов от делегатов, представляющих их в UI; возможность динамической адаптации рабочей очереди к текущей загрузке системы; упрощение разработки и многие другие.

Библиотека ThreadWeaver была разработана для удовлетворения потребностей разработчиков графических программ, управляемых событиями, но в итоге она получилась более универсальной. Так как графические программы находятся под управлением центрального цикла обработки событий, они не могут выполнять продолжительные операции в своем главном потоке, так как это приведет к блокировке пользовательского интерфейса до завершения операции. В некоторых оконных средах пользовательский интерфейс может отображаться только из главного потока или сама оконная система является многопоточной. Таким образом, естественным способом реализации межплатформенных графических приложений, быстро реагирующих на действия пользователя, является выполнение всей обработки в рабо-

чих потоках и обновление пользовательского интерфейса из главного потока по мере необходимости. Как ни странно, необходимость параллелизма в пользовательских интерфейсах редко бывает настолько очевидной, насколько она того заслуживает, хотя еще много лет назад о его важности говорилось для OS/2, Windows NT и Solaris. Многопоточное программирование отличается большей сложностью; разработчик должен лучше понимать, как работает написанный им код. Кроме того, в теме многопоточности хорошо разбираются программные архитекторы и проектировщики, но не специалисты по сопровождению и менее опытные программисты. Вдобавок некоторые разработчики полагают, что многие операции достаточно быстры для синхронного выполнения – даже чтение из смонтированных файловых систем, которое на два порядка медленнее любых вычислений на процессоре. Такие ошибки наглядно проявляются только в особых обстоятельствах – например, если система выполняет большие объемы ввода/вывода, если смонтированная файловая система переведена в ждущий режим для экономии питания, или – кто бы мог подумать! – файловая система находится в сети.

В следующем разделе описывается архитектура библиотеки и ее основополагающие концепции. В конце главы будет рассказано, как библиотека попала в KDE 4.

Знакомство с ThreadWeaver, или насколько сложной бывает загрузка файла?

Чтобы программисты пользовались средствами многопоточности, они должны быть удобными и доступными. Рассмотрим типичный пример выполнения операции в графической программе – загрузку файла в буфер, находящийся в памяти, с последующим выводом результатов. В императивной программе, в которой все операции выполняются в блокирующем режиме, алгоритм получается несложным:

1. Убедиться в том, что файл существует и доступен для чтения.
2. Открыть файл для чтения.
3. Прочитать содержимое файла в память.
4. Обработать содержимое файла.
5. Вывести результаты.

Чтобы программа могла считаться дружественной по отношению к пользователю, достаточно после каждого шага выводить в командной строке сообщение о ходе выполнения операции (если пользователь запросил подробный вывод).

В графической программе ситуация предстает в несколько ином свете, потому что во время всех перечисленных операций экран должен обновляться, а у пользователя должна быть возможность отменить начатую операцию. Звучит невероятно, но даже в недавней документации по инструментариям GUI упоминается метод «периодической проверки событий». Идея заключается в том, чтобы периодически проверять события во время продолжительных операций действий, и обновлять или отменять их в случае необходимости. Действовать нужно очень внимательно, потому что состояние приложения может изменяться весьма неожиданным образом. Например, пользователь может попытаться закрыть программу, не подозревая о том, что программа ожидает события где-то в середине стека вызовов операции. Короче говоря, схемы, основанные на опросе событий (polling), никогда не работали особенно хорошо и в основном вышли из моды.

Правильное решение базируется на использовании программных потоков (конечно). Но без поддержки инфраструктуры многопоточные реализации часто выглядят довольно странно. Так как графические программы основаны на обработке событий, каждая из перечисленных операций начинается с события, и событие же отмечает ее завершение. В мире C++ для оповещений часто применяются сигналы. Некоторые программы выглядят примерно так:

1. Пользователь выдает команду загрузки файла. Команда активизирует метод-обработчик посредством сигнала или события.
2. Запускается операция открытия и загрузки файла, которая передает управление второму методу, сообщая о своем завершении.
3. В этом методе происходит обработка данных, после чего управление передается третьему методу.
4. Последний метод выводит результаты.

Весь этот каскад методов-обработчиков плохо отслеживает состояние операции и повышает вероятность ошибок. Кроме того, он проявляет недостаточное разделение операций и представления. Тем не менее, подобная реализация применяется во многих графических приложениях.

В подобных ситуациях на помощь приходит ThreadWeaver. При использовании заданий реализация выглядит примерно так:

1. Пользователь выдает команду загрузки файла. Команда активизирует метод-обработчик посредством сигнала или события.
2. В методе-обработчике программист создает последовательность заданий (последовательность представляет собой контейнер, выполняющий задания в порядке их добавления). Он добавляет в после-

довательность задание для загрузки файла и другое задание для обработки его содержимого. Объект последовательности сам по себе является заданием и отправляет сигнал после завершения всех содержащихся в нем заданий. До этого момента никаких реальных действий еще не выполнялось; программист всего лишь объявил, что и в каком порядке необходимо сделать. После создания всей последовательности пользователь ставит ее в очередь заданий уровня приложения (синглетный экземпляр с отложенной инициализацией). Последовательность автоматически выполняется рабочими потоками.

3. При получении от последовательности сигнала `done()` данные готовы к выводу.

В такой схеме очевидны два аспекта. Во-первых, все отдельные этапы объявляются разом, а затем отправляются на выполнение. Уже одно это существенно облегчает задачу программистов GUI, потому что данная операция сопряжена с невысокими затратами и может легко выполняться в обработчике события. Во-вторых, простое правило о том, что поток работы с очередью может обращаться к данным задания только после того, как они будут подготовлены, позволяет избежать многих стандартных проблем синхронизации. Таким образом, доступ к данным осуществляется последовательно, но в кооперативном режиме. Если программист захочет выводить информацию о ходе выполнения, последовательность выдает сигналы после обработки каждого отдельного задания (сигналы в Qt могут передаваться между потоками). Графический интерфейс по-прежнему оперативно реагирует на действия пользователя; он может выводить задания из очереди или запрашивать отмену выполнения.

Подобная схема значительно упрощает реализацию операций ввода/вывода, поэтому библиотека ThreadWeaver была быстро принята сообществом программистов. Она позволяла решать стандартные задачи простым, удобным способом.

Основные концепции и возможности

В предыдущем примере были упомянуты последовательности заданий. Давайте посмотрим, какие еще конструкции используются в библиотеке.

Последовательности представляют собой специализированную разновидность коллекций заданий. Коллекции заданий представляют собой контейнеры, которые помещают в очередь наборы заданий за одну атомарную операцию и оповещают программу о всем наборе. Коллекции заданий являются составными объектами в том отношении, что они

сами реализуются как классы заданий. В ThreadWeaver существует только одна операция постановки в очередь: ей передается указатель на Job. Составные задания позволяют ограничиться минимальным API для работы с очередью.

Механизм зависимостей обеспечивает выполнение заданий, содержащихся в последовательностях, в правильном порядке. Если между двумя заданиями существует зависимость, это означает, что зависимое задание может быть выполнено только после завершения того задания, от которого оно зависит. Так как зависимости могут объявляться по схеме *m:n*, практически любые схемы логики зависимости операций (представленные направленными графами, так как повторение заданий запрещено) могут быть смоделированы на декларативном уровне. Если граф выполнения остается направленным, задания даже могут ставить в очередь другие задания в ходе своего выполнения. Типичный пример – генерирование веб-страницы, в ходе которого якорные элементы обрабатываются только после обработки текста самого документа HTML. В очередь добавляются задания для загрузки и подготовки всех связанных элементов, а последнее задание, зависящее от этих подготовительных заданий, генерирует страницу для отображения. Специальные примитивы синхронизации при этом не нужны.

Именно зависимости отличают систему планирования типа ThreadWeaver от обычных инструментов параллельного выполнения. Они избавляют программиста от необходимости думать о том, как лучше всего распределять отдельные субоперации между потоками. Даже с такими современными концепциями, как фьючерсы (futures), программисту обычно приходится определять порядок операций. При использовании ThreadWeaver рабочие потоки выполняют все возможные задания, не имеющие неразрешенных зависимостей. Поскольку выполнение графов с параллельной логикой в принципе имеет недетерминированную природу, крайне маловероятно, что определенный вручную порядок будет достаточно гибким, чтобы стать самым эффективным. Планировщик гораздо лучше адаптируется к ситуации. Возможно, ученые в области компьютерных технологий не согласятся с этим утверждением, но экономисты, более привычные к анализу стохастических систем, часто поддерживают его.

Для управления порядком выполнения также могут использоваться приоритеты. Система приоритетов весьма проста: свободному рабочему потоку на выполнение первыми передаются задания, которым назначены более высокие целочисленные приоритеты. Поскольку базовый класс задания реализован таким образом, что для него можно писать декораторы, внешнее изменение приоритета задания можно реализовать написанием декоратора, который повышает приоритет без

изменения реализации задания. Как будет показано ниже, комбинация приоритетов и зависимостей порой приводит к интересным результатам.

Вместо того чтобы полагаться на непосредственные реализации поведения очередей, ThreadWeaver использует политики. Политика очереди не оказывает прямого влияния на то, как и когда будет выполняться то или иное задание. Вместо этого она влияет на порядок выборки заданий из очереди рабочими потоками. В поставку ThreadWeaver включены две стандартные реализации политик: первая – зависимости, о которых говорилось ранее. Вторая – ресурсные ограничения. С помощью ресурсных ограничений можно объявить, что из некоторого подмножества всех созданных заданий (например, высокозатратных заданий ввода/вывода с локальной файловой системой) одновременно может выполняться только определенное подмножество заданий. Без такого инструмента в некоторых подсистемах будут регулярно возникать перегрузки. Ресурсные ограничения отчасти напоминают семафоры в традиционных многопоточных схемах, не считая того, что они не могут блокировать вызывающий поток, а вместо этого просто отмечают, что задание на данный момент выполняться еще не может. После проверки задания поток может перейти к попытке выполнения другого задания.

Политики очередей назначаются заданиям, причем один объект политики может быть присвоен нескольким заданиям сразу. Любое задание может находиться под управлением произвольной комбинации доступных политик. Создание специализированных классов заданий с определенными политиками посредством наследованием от базового класса задания не обеспечило бы такой гибкости. Кроме того, при таком подходе объекты заданий, не нуждающиеся в дополнительных политиках, не страдают от возможных затрат ресурсов, связанных с проверкой политик.

Декларативная многопоточность на примере просмотра миниатюр

Следующий пример показывает, как разные концепции ThreadWeaver работают в сочетании друг с другом. Задания, композиции заданий, ресурсные ограничения, приоритеты и зависимости используются в нем для отображения миниатюрных изображений в графической программе. Для начала давайте разберемся, какие операции необходимы для реализации этой функции, как они зависят друг от друга и в какой форме должны выводиться результаты. Приведенный пример входит в исходный код ThreadWeaver.

Предполагается, что загрузка миниатюры для предварительного просмотра цифровой фотографии состоит из трех операций: загрузки данных из файла на диске, преобразования необработанных данных в представление изображения без изменения размера и последующего масштабирования по размерам миниатюры. Можно сказать, что второй и третий этапы можно объединить, но это (а) не является целью нашего упражнения и (б) вводит дополнительные ограничения на использование только тех графических форматов, для которых драйверы поддерживают масштабирование при загрузке. Также предполагается, что файлы хранятся на жестком диске. Поскольку обработка каждого файла не влияет и не зависит от обработки других файлов, все файлы могут обрабатываться параллельно. Три операции по обработке одного файла должны выполняться последовательно.

Но это еще не все. Так как речь идет о графическом пользовательском интерфейсе, необходимо учитывать ожидания пользователя. Вероятно, пользователь захочет получать визуальную обратную связь, которая помогает ему судить о ходе операции. Миниатюры изображений должны отображаться сразу же после того, как они станут доступными, а для вывода информации о ходе операции должен использоваться индикатор, который своевременно выводит полученную информацию. Пользователь также ожидает, что программа не парализует работу его компьютера – например, слишком интенсивными операциями ввода/вывода.

Для решения задачи могут использоваться разные инструменты ThreadWeaver. Прежде всего, обработка отдельного файла представляет собой последовательность реализаций трех заданий. Задания имеют относительно общий характер и могут быть позаимствованы из инструментария готовых классов заданий, доступных приложениям. Будут использоваться следующие классы заданий (имена классов соответствуют именам из исходного кода):

- `FileLoaderJob` загружает файл из файловой системы в байтовый массив, находящийся в памяти.
- `QImageLoaderJob` преобразует физические данные изображения в типичное представление графики в приложениях Qt (в результате чего приложение получает доступ ко всем доступным декодерам – находящимся в инфраструктуре или зарегистрированным приложением).
- `ComputeThumbNailJob` масштабирует изображение до заданного размера.

Все эти задания включаются в последовательности `JobSequence`, а каждая последовательность включается в `JobCollection`. Составная природа все классов коллекций позволяет использовать реализации, очень

близко представляющие исходную задачу, а следовательно, несколько более естественные и канонические для программиста.

Так решается первая часть задачи – параллельная обработка разных изображений. Однако она легко может привести к другим проблемам. При таком объявлении задачи в очереди ThreadWeaver ничто не помешает ей загрузить все файлы одновременно и только потом приступить к обработке изображений. Хотя это маловероятно, мы еще не сообщили системе никакой информации, которая могла бы предотвратить подобную ситуацию. Чтобы ограничить количество одновременно выполняемых загрузчиков файлов, необходимо воспользоваться ресурсными ограничениями. Код выглядит примерно так:

```
#include "ResourceRestrictionPolicy.h"

...

static QueuePolicy* resourceRestriction()
{
    static ResourceRestrictionPolicy policy( 4 );
    return &policy;
}
```

Загрузчики файлов просто применяют политику в своем конструкторе или при использовании параметризованных классов – при их создании:

```
fileloader->assignQueuePolicy( resourceRestriction() );
```

Но и этот вариант еще не определяет нужного нам порядка выполнения заданий. Теперь очередь сможет запустить не более четырех загрузчиков файлов одновременно, но все равно может загрузить все файлы, а затем приступить к построению миниатюр (хотя еще раз подчеркнем, что это поведение крайне маловероятно). Для решения проблемы нам понадобится еще один инструмент и немного нестандартного мышления; в игру вступают приоритеты. В переводе на язык ThreadWeaver проблема заключается в том, что задания загрузчиков файлов обладают наименьшим приоритетом, но должны выполняться первыми; задания преобразования графики приоритетнее загрузчиков файлов, но их запуск возможен только после завершения соответствующих загрузчиков файлов; наконец, задания масштабирования обладают наивысшим приоритетом даже несмотря на то, что они зависят от двух других фаз обработки. Так как три задания уже находятся в последовательности, которая обеспечит их выполнение в правильном порядке для каждого изображения, назначение приоритета 1 загрузчикам файлов, приоритета 2 загрузчикам графики и приоритета 3 построителям миниатюр решит проблему. В двух словах, очередь

теперь будет завершать одну миниатюру как можно скорее, но не будет отвлекаться на загрузку изображений при наличии слотов для загрузки файлов. Так как проблема главным образом связана с вводом/выводом, это означает, что общее время вывода всех изображений немного превышает время их загрузки с жесткого диска (если забыть об остальных возможных факторах, например, RAW-изображениях с чрезвычайно высоким разрешением). В любом последовательном решении результат, скорее всего, будет заметно хуже.

Возможно, описание решения выглядит слишком сложно и его стоит пояснить небольшим фрагментом кода. После того как пользователь выберет пару сотен изображений для обработки, задания генерируются следующим образом:

```
m_weaver->suspend();
for (int index = 0; index < files.size(); ++index)
{
    SMIVItem *item = new SMIVItem ( m_weaver, files.at(index ), this );
    connect ( item,  SIGNAL( thumbReady(SMIVItem* ) ),
              SLOT ( slotThumbReady( SMIVItem* ) ) );
}
m_startTime.start();
m_weaver->resume();
```

Чтобы информация о ходе выполнения операции отображалась корректно, обработка приостанавливается до добавления всех заданий. При каждом завершении последовательности объект `item` выдает сигнал на обновление представления. Для каждого выбранного файла создается специальный элемент, который, в свою очередь, создает объекты заданий для обработки одного файла:

```
m_fileloader = new FileLoaderJob ( fi.absoluteFilePath(),  this );
m_fileloader->assignQueuePolicy( resourceRestriction() );
m_imageloader = new QImageLoaderJob ( m_fileloader,  this );
m_thumb = new ComputeThumbNailJob ( m_imageloader,  this );
m_sequence->addJob ( m_fileloader );
m_sequence->addJob ( m_imageloader );
m_sequence->addJob ( m_thumb );
weaver->enqueue ( m_sequence );
```

Приоритеты являются виртуальными свойствами объектов заданий и задаются в этом фрагменте. Важно помнить, что все объекты настроены таким образом, чтобы обработка начиналась только после помещения в очередь, а в нашем случае – только после ее явного возобновления. Таким образом, вся операция по созданию последовательностей и заданий занимает минимальное время, а программа возвращает

управление пользователю практически немедленно. Изображение обновляется сразу же после появления готовой миниатюры.

От параллелизма к планированию: как систематически реализовать ожидаемое поведение

Предыдущие примеры показывают, как полный анализ задачи помогает решить ее (надеюсь, это не вызовет ни у кого удивления). Чтобы параллелизм действительно помогал строить лучшие программы, недостаточно предоставить инструмент для распределения функциональности по программным потокам. Принципиальным различием является планирование: необходимо иметь возможность сообщить программе, какие операции она должна выполнить и в каком порядке. Такой подход напоминает давно забытые уроки программирования на PROLOG, а иногда требует аналогичного менталитета. Когда вы сможете в достаточной степени приспособиться к нему, результат оправдает потраченные усилия.

Мы еще не обсудили одно из важных архитектурных решений центрального класса `Weaver`. Пользователи API `Weaver` делятся на две очень сильно различающиеся группы. Внутренние объекты `Thread` обращаются к нему за заданиями для выполнения, а программисты – для управления параллельными операциями. Чтобы по возможности сократить размер общедоступного API, была применена комбинация паттернов «декоратор» и «фасад», которая ограничивает общедоступный API функциями, предназначенными для использования прикладными программистами. Дальнейшее разделение внутренней реализации и API было достигнуто посредством применения идиомы PIMPL, которая обычно применяется ко всем API проекта KDE.

Безумная идея

Как упоминалось ранее, в начале разработки ThreadWeaver реализовать все идеи было невозможно. Одно серьезное препятствие оказалось непреодолимым: применение нетривиальных средств неявного совместного доступа, включающих подсчет ссылок, в библиотеке Qt. Поскольку неявный совместный доступ не обладал потоковой безопасностью, передача каждого объекта POD (Plain Old Data) становилась точкой синхронизации. Автор решил, что такое решение будет непрактично с точки зрения пользователей и по этой причине не рекомендовал использовать прототип, разработанный с использованием Qt 3, в среде реальной эксплуатации. Разработчики пакета KDEPIM (те же, которые сейчас разрабатывают Akonadi) решили, что им виднее, и немедленно импортировали предварительную версию ThreadWeaver в KMail, где

она используется до сегодняшнего дня. Разработчики KMail ранее столкнулись со многими проблемами, которые библиотека ThreadWeaver обещала решить, и с энтузиазмом ухватились за нее. При этом они заранее смирились со всеми недостатками, упомянутыми автором, и даже наперекор его недвусмысленным предупреждениям.

Факт активного использования несовершенной версии библиотеки в KDE послужил стимулом для ее быстрого портирования на версию Qt 4, когда последняя появилась в бета-версиях. Таким образом, она довольно рано появилась в цикле разработки KDE 4, хотя и в виде вторичного модуля, еще не ставшего частью *KDELibs*. За прошедшие два года автор провел ряд презентаций библиотеки, на которых описывал API, который в процессе усовершенствования постепенно упрощался и становился все более полным. Можно сказать, что у нас появилось решение, которому требовалась достойная задача. Большинство разработчиков, трудившихся над KDE, не сразу осознало, что библиотека имела не только академическую ценность, но и могла значительно улучшить их программный код – если не полениться сделать шаг назад и заново продумать некоторые из архитектурных структур. Развитие библиотеки не было обусловлено конкретными потребностями группы разработчиков; библиотекой занимался один человек, который верил в растущую актуальность проблемы и важность создания хорошего решения для платформы KDE 4. После конференции Akademy 2005 в Малаге (Испания) библиотека ThreadWeaver стала использоваться во все большем количестве программ, включая KOffice и KDevelop. Это создало импульс, достаточный для интеграции ThreadWeaver в основной набор библиотек KDE 4.

На примере ThreadWeaver мы видим, как альтернативное решение проблемы созрело до критического уровня, а автор и сообщество потенциальных пользователей согласились с тем, что разработчикам пора использовать его в своих проектах, после чего решение быстро стало одним из краеугольных камней KDE 4. После этого изменилось и отношение членов сообщества: слабая заинтересованность превратилась в признательность и понимание потраченных усилий. Эта ситуация показывает, насколько эффективным может быть сообщество при принятии технических решений и как может меняться его позиция, когда метод оправдывает себя на практике. Нет сомнений в том, что успех библиотеки ThreadWeaver во многом обусловлен теми тремя или четырьмя годами, в течение которых она «притиралась» к проекту KDE до ее включения. Кстати, сюда относится и самовольное преждевременное ее принятие разработчиками KMail. Не приходится сомневаться в том, что приложения, написанные для KDE 4, намного лучше используют многопоточность (а, следовательно, и улучшают впечатле-

ния своих пользователей от работы) благодаря тому, что библиотека успешно справилась со своей задачей.

Расширение ThreadWeaver будет в основном осуществляться добавлением компонентов GUI для графического представления операций с очередями и включением большего количества готовых классов заданий. Другим возможным направлением является интеграция с механизмами IPC операционных систем (например, для установления ресурсных ограничений, глобальных по отношению к хосту), но развитие в этом направлении тормозится требованиями кросс-платформенности. Механизмы, используемые в разных операционных системах, очень сильно отличаются друг от друга. С широким распространением линейки KDE 4 это обстоятельство стало очевидным для массовой аудитории. Поскольку библиотека ThreadWeaver не привязана к KDE, вопрос о том, куда идти дальше (*freedesktop.org*?), остается открытым. А пока основные усилия направляются на то, чтобы предоставить разработчикам приложений и сред рабочих столов надежный планировщик для параллельного выполнения.

V

Языки и архитектура

13

Программные архитектуры: объектно-ориентированные и функциональные

Бертран Мейер

Одним из аргументов в пользу функционального программирования считается улучшение модульности архитектуры. Анализ публикаций, поддерживающих этот подход (прежде всего на примере инфраструктуры для оформления финансовых контрактов), поможет нам оценить его сильные и слабые функционального программирования и сравнить его с объектно-ориентированным проектированием. Общий вывод будет таким: объектно-ориентированное проектирование (особенно в его современной форме с поддержкой высокоуровневых функциональных объектов, или «агентов») *замещает* функциональный подход, сохраняя большинство из его преимуществ, но при этом предоставляет высокоуровневые абстракции, лучше приспособленные для возможного расширения и повторного использования.

Обзор

«Красота» как лозунг программной архитектуры не является сугубо субъективным показателем. Существуют четкие критерии (Meyer, 1997):

Надежность

Оказывает ли архитектура положительное влияние на правильность и устойчивость работы программного продукта?

Расширяемость

Насколько легко она адаптируется к изменениям?

Универсальность

Имеет ли решение общий характер – или еще лучше, возможно ли преобразовать его в *компонент*, который может быстро и напрямую подключаться к новому приложению?

Успех объектных технологий в значительной степени обусловлен тем заметным положительным влиянием, которые они могут оказать (при правильном применении на методологическом уровне, а не простым выбором объектно-ориентированного языка программирования) на надежность, возможность расширения и универсальность создаваемых программ.

Методология *функционального программирования* старше объектно-ориентированного подхода. Ее истоки уходят к языку Lisp, который существует уже почти 50 лет. Те счастливчики, которые изучали ее на ранней стадии, всегда будут помнить ее, словно первый поцелуй, сладость и предчувствие еще больших наслаждений. В последнее время функциональное программирование переживает новый подъем в связи с появлением таких языков, как Scheme, Haskell, OCaml и F#, сложных систем типизации и нетривиальных языковых механизмов (таких как монады). Иногда функциональное программирование даже представляется как следующий шаг после объектно-ориентированных методологий. В данной главе мы сравним эти два подхода по критериям программной архитектуры, указанным выше. Оказывается, объектно-ориентированная архитектура, особенно обогащенная новейшими достижениями (такими, как *агенты* в терминологии Eiffel, также называемыми «замыканиями» или «делегатами» в других языках), превосходит функциональное программирование, сохраняя его архитектурные преимущества и исправляя недостатки.

Для адекватной оценки полученных результатов важно понимать как ограничения анализа, так и аргументы, опровергающие некоторые из них. Ограничения:

Малый объем выборки

Анализ в основном базируется на двух примерах функционального проектирования. Это обстоятельство может поставить под сомнение общий характер выводов.

Малая детализация

Примеры, представленные в главе, позаимствованы из статьи (Peyton Jones et al., 2000) и презентации PowerPoint (Eber et al., 2001), которые в дальнейшем будут именоваться «статьей» и «презентацией» (с дополнениями из классической статьи по функциональному программированию [Hughes, 1989]), в презентации могут быть опущены некоторые подробности, присутствующие в более подробном документе).

Ограниченность подхода

Мы ограничимся рассмотрением аспектов модульности. К сильным сторонам функционального программирования также относятся и другие критерии – например, элегантность декларативного подхода.

Субъективизм экспериментатора

Автор настоящей главы в течение долгого времени являлся сторонником объектных технологий и вносил активный вклад в их развитие.

Возможная критика отчасти компенсируется следующими обстоятельствами.

- Примеры функционального программирования позаимствованы из реальной жизни, а именно из практики компании, бизнес которой основан на применении методов функционального программирования. Приведенный пример – определение сложных средств для описания финансовых контрактов – отражает сложные проблемы, с которыми сталкивается финансовая отрасль и которые недостаточно хорошо решаются текущим инструментарием (по утверждениям автора, являющегося экспертом в этой отрасли). Возникает предположение, что такая ситуация типична для нынешнего состояния дел (первый пример этой главы – описание пудингов – имеет чисто академический характер и приводится исключительно в учебных целях).
- Один из авторов статьи (С. Пейтон Джонс), также упоминаемый в презентации как соавтор теоретической работы, является ведущим проектировщиком языка Haskell и одной из самых заметных фигур в области функционального программирования, что придает его утверждениям достоверность. Статья, использованная в качест-

ве дополнительного примера в разделе «Оценка модульности функциональных решений», считается в высшей мере авторитетной и была написана одним из ведущих экспертов сообщества функционального программирования (Дж. Хьюз).

- Невзирая на все замечания, описанные в этих документах решения весьма элегантны и, несомненно, появились в результате серьезных размышлений.
- В примерах не задействована концепция изменяемого *состояния*, благоприятствующая применению объектно-ориентированных языков программирования.

Следует заметить, что такие механизмы, как агенты, являющиеся исключительно важными компонентами объектно-ориентированных решений, явно вдохновлены идеями функционального программирования. Таким образом, наше заключение ни в коей мере не отрицает вклада функциональной школы, а всего лишь отмечает, что объектно-ориентированный (ОО) стиль лучше подходит для определения общей архитектуры надежных, расширяемых и универсальных программ, а среди структурных блоков таких архитектур могут присутствовать комбинации как методов ОО, так функциональных методов.

Еще несколько замечаний по поводу следующего обсуждения.

- Объектная технология, использованная в примерах, представлена на языке Eiffel. Мы не пытались анализировать, что произойдет при *удалении* таких механизмов, как множественное наследование (не поддерживаемое в Java и C#), параметризация (отсутствующая в ранних версиях этих языков), контракты (отсутствующие за пределами Eiffel, кроме JML and Spec#) и агенты с их аналогами (отсутствуют в Java) или при *добавлении* таких механизмов, как перегрузка и статические функции, противоречащих изначальной простоте объектно-ориентированного подхода.
- Темой обсуждения являются архитектура и проектирование. Несмотря на свое название, функциональное программирование (как и объектные технологии) имеет отношение к этим задачам и не ограничивается «программированием» в ограниченном смысле, т. е. реализацией. Методология Eiffel явным образом вводит континуум из спецификации в проектирование и реализацию через концепцию плавной разработки. Мы не будем сколько-нибудь подробно обсуждать аспекты обеих методологий, относящиеся к реализации.
- Также для практического программирования важны аспекты выразительности и удобства записи. Они принимаются во внимание в той степени, в которой влияют на ключевые критерии архитектуры и проектирования. Однако в общем и целом обсуждение будет

относиться не к синтаксической форме, а к семантическому содержанию.

И еще два предварительных замечания. Первое касается терминологии: по умолчанию термин «контракт» будет использоваться для обозначения финансовых контрактов, относящихся к предметной области статьи и презентации; не путайте их с концепцией контрактов в программировании (Meyer, 1997), относящейся к элементам спецификации (предусловия, постусловия, инварианты). В случае возможной неоднозначности будут использоваться термины *финансовые контракты* и *программные контракты*.

Второе замечание больше напоминает самооправдание: когда во второй половине обсуждение переходит в объектно-ориентированную область, текст содержит больше ссылок и цитат из предыдущих публикацией автора, чем допускают приличия. Дело в том, что широкое распространение объектных технологий сопровождалось утратой некоторых неочевидных, но (по нашему мнению) критически важных принципов, как, например, разделение команд и запросов (см. раздел «Проблема состояния» далее в этой главе); по этой причине короткие напоминания необходимы. За полными обоснованиями обращайтесь по ссылкам.

Примеры

Общая цель статьи и презентации заключается в создании удобного механизма описания финансовых контрактов и работы с ними – особенно с современными финансовыми инструментами, которые бывают очень сложными, как в следующем примере из презентации (в числовых значениях доносятся ностальгические отзвуки тех времен, когда основные валюты находились в несколько иных соотношениях):

«Против обещания оплаты USD 2.00 27 декабря (по цене опциона) владелец имеет право 4 декабря выбрать одно из двух:

- получить USD 1.95 29 декабря или
- иметь право 11 декабря выбрать одно из двух:
  - получить EUR 2.20 28 декабря или
  - иметь право 18 декабря выбрать одно из двух:
    - получить GBP 1.20 30 декабря или
    - немедленно доплатить EUR 1.00 и получить EUR 3.20 29 декабря».

Приведенные в этом разделе цитаты в кавычках взяты непосредственно из презентации или статьи. Элементы, не заключенные в кавычки, – наши интерпретации и комментарии.

Презентация начинается с учебного примера, который помогает проиллюстрировать материал: контракты заменяются *пудингами*. По точному описанию пудинга повар должен иметь возможность «вычислить содержание сахара», «оценить время приготовления» и получить «инструкции по приготовлению». «Плохое решение» выглядит так:

- «Перечислить все разновидности пудингов (со взбитыми сливками, лимонный, голландский яблочный, рождественский).
- Для каждого пудинга записать содержание сахара, время приготовления, инструкции и т. д.».

Хотя презентация не объясняет, почему этот подход плох, причины легко понятны. Набор конкретных рецептов не обладает универсальностью, так как в нем не используется тот факт, что разные виды пудингов могут состоять из одних базовых компонентов. Он не может расширяться, поскольку любое изменение компонента потребует переработки всех зависящих от него рецептов.

Пудинг – всего лишь метафора для того, что представляет для нас реальный интерес (т. е. контрактов), но поскольку их описание не требует никаких специальных знаний, мы продолжим использовать этот пример. «Хорошее решение» заключается в следующем:

- «Определить небольшой набор „комбинаторов“.
- Определить все пудинги в терминах комбинаторов.
- Вычислить содержимое сахара на основании комбинаторов».

Комбинатор представляет собой оператор, создающий составной объект из нескольких сходных объектов. Позаимствованное из презентации дерево на рис. 13.1 показывает, какие комбинаторы могут использоваться для описания пудингов.

Примечание

Мы разделяем тревогу читателя по поводу неаппетитной природы этого примера – особенно для автора, живущего в Париже. В оправдание можно сказать только то, что презентация была рассчитана на иностранную аудиторию, которая (наряду с незнанием метрической системы) обычно обладает весьма тривиальными кулинарными вкусами. Далее будем считать, что неразборчивость в выборе десерта не предполагает неразборчивости в выборе языка и архитектурных парадигм.

Нелистовые узлы дерева представляют комбинаторы, применяемые к поддеревьям. Например, «Взять» – комбинатор, получающий два аргумента: ингредиент («Сливки» слева, «Апельсины» справа) и количество («1 пинта» и «6»). Результатом применения комбинатора, обозна-

Рис. 13.1. *Ингредиенты и комбинаторы, описывающие рецепт пудинга*

ченного узлом дерева, является компонент пудинга (или весь пудинг), состоящий из заданного количества экземпляров компонента.

Такая структура также может быть выражена в текстовом виде, с применением «мини-языка предметной области» (DSL, Domain Specific Language) «для описания пудингов» (жирным шрифтом выделяются операторы):

```
"салат              = положить_сверху украшение основная_часть"
"украшение          = взбить (взять 1 пинта сливки)
основная_часть      = смешать яблочная_часть апельсиновая_часть
яблочная_часть      = нарезать (взять 3 яблоки)
апельсиновая_часть  = по желанию (взять 6 апельсины)"
```

В этой записи используется анонимная, но весьма типичная разновидность записи функционального программирования, в которой применение функции записывается аргументами функции (например, `плюс a b` для применения операции `плюс` к `a` и `b`), а скобки используются только для группировки.

При подобном подходе такая операция, как определение содержания сахара (`S`), задается посредством анализа комбинаторов (по аналогии с определением математической функции для рекурсивных объектов с сохранением той же рекурсивной структуры):

```
"S (положить_наверх p1 p2)    = S (p1) + S (p2)
S (взбить p)                  = S (p)
S (взять q i)                 = q * S(i)
и т. д."
```

Из «и т. д.» совершенно неясно, как операторы типа `S` должны поступать с комбинатором `по_желанию`; должен существовать какой-то способ определить, содержит ли конкретная заготовка необязательный компонент, или нет. Но даже если забыть об этом обстоятельстве, описанный подход обеспечивает все преимущества, перечисленные в презентации:

- «При определении нового рецепта мы можем вычислить содержание сахара без дополнительной работы.
- Модификация `S` потребуется только при добавлении новых комбинаторов или новых ингредиентов».

Конечно, нашей настоящей целью являются не пудинги, а контракты. В презентации приводится общая схема, но статья содержит более подробную информацию. Она базируется на тех же идеях, примененных к более интересному набору элементов, комбинаторов и операций.

Элементами являются финансовые контракты, даты и наблюдаемые условия (например, курс обмена на определенную дату). Примеры простейших контрактов: `zero` (возможна покупка в любое время, без прав и обязательств) и `one (c)` для валюты `c` (немедленная выплата владельцу одной единицы `c`).

Примеры использования *комбинаторов* в контрактах: `or` (приобретение контракта `(or c1, c2)` означает приобретение либо `c1`, либо `c2`, а срок действия контракта прекращается с истечением срока действия `c1` и `c2`); `anytime` (контракт `(anytime c)` может быть приобретен в любой момент до истечения срока действия `c`, а его срок действия истекает одновременно с `c`); `truncate` (контракт `(truncate t c)` эквивалентен `c` за исключением того, что срок его действия истекает в более раннюю из дат `t` и истечения срока действия `c`); `get` (или `(get c)`, означает приобретение *c* с истечением срока действия). В статье перечислено около дюжины подобных базовых комбинаторов для контрактов и другие возможные варианты для дат и условий. Это позволяет определять сложные финансовые инструменты (скажем, «европейский опцион») простым способом:

```
european t u   = get (truncate t (or u zero))
```

Операции включают дату истечения срока действия контракта и (в конечном итоге самое важная практическая польза от нашего моделирования) его оценочную серию – проиндексированную по времени последовательность ожидаемых значений стоимости контракта. Как и в случае с содержанием сахара в пудинге, функции определяются посредством анализа базовых конструкторов. Несколько примеров с применением перечисленных выше базовых элементов и комбинаторов к операции `H`, обозначающей функцию даты истечения срока действия («горизонт»):

```
H (zero)          = ∞ – специальное значение с особыми свойствами
H (or c1 c2)      = max (H (c1), H (c2))
H (anytime c)     = H (c)
H (truncate t c)  = min (t, H (c))
H (get c)         = H (c)
```

Правила вычисления оценочных серий имеют аналогичную структуру, хотя правая сторона выражений, естественно, получается более сложной и включает различные финансовые и числовые операции. За дополнительными примерами применения комбинаторов и идей функционального программирования в финансовых приложениях обращайтесь к Frankau [2008].

Оценка модульности функциональных решений

Описание из предыдущего раздела, хотя в нем опущены многие тонкости презентации и особенно статьи, может послужить основой для обсуждения архитектурных особенностей функционального подхода и их сравнения с объектно-ориентированным представлением. Далее мы будем свободно переключаться между примерами с пудингами (они позволяют немедленно понять идею) и финансовыми контрактами (типичными для реальных приложений).

Критерии расширяемости

Как указано в презентации, прямая польза такой архитектуры заключается в простоте добавления нового комбинатора: «При определении нового рецепта мы можем вычислить содержание сахара без дополнительной работы». Однако это свойство вряд ли можно назвать следствием применения функционального подхода. В нашем рассмотрении было введено понятие комбинатора, создающего пудинг или его отдельные компоненты (или контракты) из других компонентов, которые в свою очередь могут быть как атомарными, так и результатами применения комбинаторов к более элементарным компонентам.

Из статьи и презентации может показаться, что в них представлена новая идея представления финансовых контрактов, но в действительности этот подход не отличается новизной. Если перевести его в область проектирования графических интерфейсов, «плохое решение», отвергнутое в начале презентации (перечисление всех типов пудингов, отдельное вычисление содержания сахара для каждого из них и т. д.) соответствует детальной проработке каждого экрана интерактивного приложения и записи соответствующих операций – вывода, перемещения, изменения размеров, сокрытия. Никто так не поступает; любая среда проектирования графических интерфейсов предоставляет атомарные

элементы (такие как кнопки и команды меню) и операции для их рекурсивного объединения в окна, меню и другие контейнеры для формирования сложного интерфейса. По аналогии с тем, как комбинаторы в примере с пудингами определяли содержание сахара и количество калорий по ингредиентам, а комбинаторы в примере с контрактами определяли горизонт и оценочную серию сложного контракта в контексте его составляющих, операции вывода, перемещения, изменения размеров и сокрытия сложной фигуры сводятся к рекурсивному применению этих операций к компонентам. Библиотека EiffelVision (Eiffel Software: документация EiffelVision) применяет композиционные принципы особенно систематическим образом, но такой подход вряд ли можно назвать уникальным. Таким образом, статья всего лишь применяет известную методологию в новой прикладной области финансовых контактов. Однако методология не требует обязательной реализации на базе функционального программирования; подойдет любая инфраструктура с механизмом передачи управления и рекурсией.

Интересные проблемы модульности возникают не при применении существующих комбинаторов к компонентам существующих типов, а при изменении типов комбинаторов и компонентов. В презентации говорится: «Модификация S (комбинатор сахара) потребуется только при добавлении новых комбинаторов или новых ингредиентов». Вопрос в том, насколько разрушительными будут такие изменения для архитектуры.

В действительности состав значимых изменений оказывается более обширным:

- Наряду с *атомарными типами* и *комбинаторами* необходимо учитывать изменения в *операциях*: добавление функции подсчета калорий для пудингов, операций задержки для контрактов, операции поворота для графических объектов.
- Кроме добавления во всех перечисленных категориях следует также учитывать возможные операции *изменения* и *удаления*, хотя для простоты мы будем и дальше ограничиваться только операциями добавления.

Оценка функциональной методологии

Структура программ в приведенном виде достаточно проста. Она состоит из определений в форме:

```
0 (a)             = b_{a,0}                [1]
0 (c (x, y, ...)  = f_{c,0} (x, y, ...)    [2]
```

для каждой операции `0`, атомарного типа `a` и комбинатора `c`. В правой части указываются соответствующие константы `b` и функции `f`. И снова для простоты атомарные типы, такие как `a`, будут рассматриваться как 0-арные комбинаторы, поэтому нам достаточно рассмотреть только форму [2]. С `t` основными комбинаторами (`положить_наверх`, `взбить`) и `f` операциями (содержание сахара, калории) потребуется `t × f` определений.

Независимо от выбора методологии эти `t × f` элементов необходимо как-то разместить. Архитектурная проблема заключается в том, как сгруппировать их по модулям для упрощения расширения и возможности повторного использования. В статье и презентации эта проблема не рассматривается. Конечно, для малых значений `t` и `f` она не критична; в этом случае все определения можно упаковать в одном модуле. При таком решении проблема расширяемости решается просто:

- Чтобы добавить комбинатор `c`, добавьте `f` определений в приведенной выше форме, по одному для каждой существующей операции.
- Чтобы добавить операцию `0`, добавьте `t` определений, по одному для каждого существующего комбинатора.

Такой подход плохо масштабируется, и в более крупных разработках систему придется делить на модули. В этом случае проблема расширяемости будет заключаться в выборе организации, при которой такие изменения влияют на минимальное количество модулей.

Даже при относительно малых `t` и `f` одномодульное решение препятствует повторному использованию. Если другой программе понадобится ограниченное подмножество операций и комбинаторов, она столкнется со стандартной дилеммой примитивной модуляризации:

Харибда

Копирование/вставка нужных фрагментов — но с риском того, что производные модули не будут обновлены в случае изменения оригинала (например, по такой прозаической причине, как исправление ошибки).

Сцилла

Импортирование всего модуля любыми доступными средствами включения модулей. В результате проект отягощается большим количеством «балласта», что усложняет обновления и может привести к конфликтам (допустим, в производном модуле определяется новый комбинатор или функция, а в следующей версии исходного модуля вводится конфликтное определение).

Эти наблюдения попутно напоминают нам, что универсальность тесно связана с расширяемостью. Интернет-критик функционального языка OCaml (Steingold 2007)[1] приводит конкретный пример:

> Поведение модуля невозможно легко изменить за его пределами. Допустим, вы используете модуль `Time`, в котором определяется метод `Time.date_of_string` для разбора базового формата ISO8601 (`"YYYYMMDD"`), но хотите использовать расширенный формат ISO8601 (`"YYYY-MM-DD"`). Не повезло: вам придется редактировать исходную функцию – переопределить ее в своем модуле вам не удастся.

По мере роста и изменения программного продукта критическую роль начинает играть другой аспект универсальности: возможность повторного использования общих свойств. Наряду с европейскими опционами в статье вводятся «американские опционы». При описании на уровне комбинаторов они имеют другие сигнатуры (`Дата → Контракт → Контракт` и `(Дата, Дата) → Контракт → Контракт`), для каждой из них приходится определять все операции по отдельности. Однако разумно предположить, что две разновидности опционов должны иметь ряд общих свойств и операций (по аналогии с группировкой пудингов по категориям). Такие группировки упрощают моделирование и деление программы на модули, с дополнительным преимуществом (при достаточном количестве общих аспектов) в виде сокращения количества необходимых определений. Однако для этого потребуется по-новому взглянуть на предметную область задачи, чтобы кроме функций выявить важнейшие *типы*.

Такое представление будет находиться на более высоком уровне абстракции. Особенно спорной может показаться фиксация на функциях и их сигнатурах. Как сказано в статье (с сохранением курсивного выделения), «*американский опцион* обладает большей гибкостью, чем *европейский*. Обычно американский опцион предоставляет возможность приобрести контракт *в любой момент времени между двумя датами* или не приобретать его вообще». Это предполагает определение по различию: либо американские опционы являются особым случаем европейских, либо оба класса являются разновидностями более общего понятия опциона. Однако при определении в виде комбинаторов они немедленно разделяются из-за лишней даты в сигнатуре. Происходящее сродни определению концепции по реализации (в математическом, а не компьютерном смысле, но все равно с потерей абстракции и общности). Выбор типов в качестве базового механизма деления про-

[1] Цитата слегка упрощена. Включение цитаты не подразумевает подтверждения других критических замечаний на этой странице.

граммы на модули, как в объектно-ориентированных решениях, поднимает систему на более высокий уровень абстракции.

Уровни модульности

Оценка функционального программирования по критериям модульности вполне законна, потому что улучшение модульности является одним из главных аргументов в пользу этого подхода. Ранее уже приводились комментарии по этому поводу из презентации; далее приводится более общее утверждение из одной из основополагающих статей функционального программирования (Hughes, 1989), в которой сказано, что при таком подходе:

> [Программы] могут делиться на модули новыми способами и, как следствие, серьезно упрощаются. В этом факте кроется ключ к силе функционального программирования – оно значительно улучшает модульность программы. Кроме того, он устанавливает цель, к которой должны стремиться функциональные программисты: уменьшение и упрощение модулей, придание им более общего характера, объединение их с использованием новых связующих элементов, которые будут описаны ниже.

«Новые связующие элементы», упоминаемые в статье Хьюза, уже встречались нам в двух описанных примерах – систематическое применение функций без состояния, включая высокоуровневые функции (комбинаторы), применяемые к другим функциям, и обширное использование списков и других рекурсивно определяемых типов, а также концепции отложенного вычисления.

При всей своей привлекательности эти методы ориентированы только на решение проблемы мелкоструктурного модульного деления. Хьюз разрабатывает функциональную версию метода Ньютона–Рафсона для вычисления квадратного корня числа `N` с отклонением `eps` и исходным приближением `a0`:

```
sqrt a0 eps N = within eps repeat (next N) a0)
```

с соответствующими комбинаторами `within`, `repeat` и `next` и сравнивает эту версию с написанной на Fortran программой, содержащей команды `goto`. Даже если не обращать внимания на некорректность сравнения (на момент публикации статьи язык Fortran уже считался древностью, а команды `goto` вышли из употребления), понятно, почему некоторые люди предпочитают решение, основанное на использовании малых функций, связанных комбинаторами, версии с циклами. Но другие предпочитают циклы, а поскольку речь идет о мелкой структуре программ, а не о крупномасштабном модульном делении, это скорее

вопрос стиля и вкуса, нежели фундаментальный архитектурный вопрос. В частности, проблемы демонстрации правильности в обоих случаях практически совпадают. Например, в приведенном в статье Хьюза определении получение первого элемента последовательности, отличающегося от предыдущего менее чем на `eps`:

```
within eps ([a:b:rest])  =  if abs (a - b) <= eps then b
else within eps [b:rest]
```

предполагает, что расстояния между соседними элементами уменьшаются, а также совершенно точно предполагает, что одно из этих расстояний будет не больше `eps`[1]. Формулировка этого свойства подразумевает наличие некоего механизма, сходного с контрактным проектированием, для связывания с функциями предусловий (тогда как в стандартных функциональных методологиях такой механизм отсутствует). Обоснование того, что гарантирует завершение в области `eps`, по сути эквивалентно обоснованию завершения соответствующего цикла в императивном решении.

В этом и предшествующих примерах ничто не относится к крупноструктурному делению на модули. В частности, лишенная состояния природа функционального программирования никак не влияет на него (ни положительно, ни отрицательно).

Преимущества функционального подхода

Из приведенных примеров видно, что у функционального подхода остаются еще четыре важных преимущества.

Первое преимущество – запись. Несомненно, привлекательность функционального программирования отчасти обусловлена компактностью определений, как в рассмотренном примере. Это сокращает количество синтаксического балласта по сравнению с объявлениями функций в типичном императивном языке. Однако это наблюдение нуждается в нескольких уточнениях:

- При рассмотрении архитектурных аспектов, как в нашем случае, запись не столь критична. Например, можно использовать функциональный стиль проектирования с императивным языком.
- Многие современные функциональные языки, такие как Haskell и OCaml, обладают сильной типизацией, вследствие чего запись неизбежно отчасти утратит компактность; например, если проекти-

[1] В цитатах из статьи Хьюза мы с согласия автора использовали современное (Haskell) обозначение списков `[a:b:rest]`, более понятное по сравнению с исходной записью вида `cons a (cons b rest)`.

ровщик не хочет полагаться на интерфейс типа (чего на стадии проектирования делать определенно не стоит), для `within` потребуются объявления типа `Double → [Double] → Double`.

- Не каждый программист будет уверенно чувствовать себя с систематической заменой многоаргументных функций функциями, возвращающими функции (в статье о финансовых контрактах это замечание проиллюстрировано сигнатурами вида `(a → b → c) → Obs a → Obs b → Obs c`). Хотя это скорее вопрос стиля, нежели фундаментальное свойство методологии, оно встречается в этой и во многих других публикациях.

Тем не менее, лаконичная запись остается положительным свойством даже на уровне проектирования и архитектуры, и из функциональных языков программирования можно почерпнуть некоторые полезные уроки.

Второе преимущество (подчеркнутое Саймоном Пейтоном Джонсом и Диомидисом Спинеллисом в комментариях к предыдущему варианту этой главы) тоже относится к записи. Речь идет об элегантности комбинаторных выражений определения объектов. В императивном объектно-ориентированном языке эквивалентом комбинаторного выражения вида

```
положить_сверху украшение основная_часть
```

будет инструкция

```
создать пудинг .украсить (украшение, основная_часть)
```

где процедура создания (конструктор) `украсить` инициализирует атрибуты `основа` и `украшение` заданными аргументами. Комбинатор здесь применяется скорее в описательной, нежели в императивной форме. Впрочем, на практике комбинаторная форма довольно часто применяется в объектно-ориентированном программировании в форме «методов-фабрик» (в отличие от явных команд создания объектов).

Два последних преимущества имеют более фундаментальную природу. Первое – способность манипулировать с операциями как с «полноценными сущностями», то есть как объектами программы (или как с данными). Lisp впервые показал, что это можно делать эффективно. В некоторых популярных языках была предусмотрена возможность передачи функций в качестве аргументов других функций, но она не считалась фундаментальным приемом проектирования, а иногда даже рассматривалась как пережиток самомодифицирующегося кода со всеми вытекающими подозрениями по его поводу. Современные функциональные языки демонстрируют мощь интерпретации высокоуровневых функциональных конструкций как обычных объектов программы

и развивают соответствующие системы типов. Эта часть функционального программирования оказывает самое непосредственное влияние на разработку популярных методологий программирования; как будет показано ниже, концепция агента, созданная на основе концепций функционального программирования, становится полезным добавлением в исходную объектно-ориентированную архитектуру.

Четвертым серьезным преимуществом функционального программирования являются отложенные (или «ленивые») вычисления: возможность описания потенциально бесконечных вычислений (хотя следует понимать, что любое конкретное выполнение этих вычислений будет конечным). В частности, оно предполагается в приведенном ранее определении `within`; это еще более очевидно в следующем определении `repeat`:

```
repeat f a  = [a : repeat f (f a)]
```

Данная запись генерирует (в традиционном синтаксисе вызова функций) бесконечную последовательность `a`, `f(a)`, `f (f(a))...`. При определении `Next N x` в виде `(x + N / x) / 2`, определение `within` остановит вычисление этой последовательности после конечного количества элементов.

Идея весьма элегантная. По поводу ее обобщенного применения в программных архитектурах необходимо сделать пару замечаний.

Во-первых, следует учитывать проблему корректности. Простота написания потенциально бесконечных программ скрывает сложности, связанные с их обязательным завершением. Мы уже видели, что `within` предполагает наличие предусловия, но это предусловие, требующее, чтобы элементы стали ниже `eps`, не может иметь гарантированного конечного вычисления на бесконечной последовательности (свойство полуразрешимости). Проектировщикам приходится прибегать к всевозможным ухищрениям, о чем свидетельствует старая задача: сколько функциональных программистов потребуется, чтобы заменить лампочку? Если помните, заранее предсказать невозможно. Если еще остались функциональные программисты, предложить одному из них заменить лампочку. Если у него не получится, попробовать другого.

Во-вторых, отложенные манипуляции с бесконечными структурами возможны в нефункциональной среде проектирования без какой-либо специальной поддержки. Подход, основанный на абстрактных типах данных (называемый также объектно-ориентированным проектированием), обеспечивает возможность такого решения. Конечные последовательности и списки в библиотеках Eiffel доступны через API, основанный на понятии «курсора» (рис. 13.2).

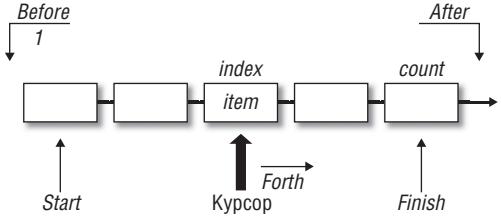


***Рис. 13.2.** Курсоры в списках Eiffel*

Перемещение курсора осуществляется командами `start` (перемещение к первому элементу), `forth` (перемещение к следующему элементу) и `finish`. Логические проверки `before` и `after` сообщают соответственно, находится ли курсор перед первым элементом или за последним. Если ни одно условие не выполняется, то `item` возвращает элемент в текущей позиции, а `index` – его индекс.

Данная спецификация легко адаптируется для бесконечных последовательностей; для этого достаточно удалить `finish` и `after` (а также `count`, количество элементов). Так выглядит спецификация отложенного (абстрактного) класса `COUNTABLE` в библиотеке Eiffel. К числу его потомков относятся классы `PRIMES`, `FIBONACCI`, `RANDOM`; каждый предоставляет собственную реализацию `start`, `forth` и `item` (а в последнем случае возможность задать значение для раскрутки генератора псевдослучайных чисел). Для получения последовательных элементов одной из этих бесконечных последовательностей достаточно применить `start`, а затем запрашивать `item` после конечного числа применений `forth`.

Этот стиль может использоваться для моделирования любой бесконечной последовательной структуры, требующей конечного вычисления. Хотя это не исчерпывает всех возможных применений отложенного вычисления, преимущество заключается в явном определении бесконечной структуры, упрощающем обеспечение корректности отложенного вычисления.

Проблема состояния

Функциональный подход стремится к прямой зависимости от свойств математических функций, для чего он отвергает предположение, неявно присутствующее в императивных методологиях, что вычислительные операции наряду с выдачей результата (как у математической функции) могут изменять состояние вычисления – глобальное состояние, или, при более модульном подходе, некоторую его часть (например, содержимое конкретного объекта).

Хотя это свойство бросается в глаза во всех описаниях функционального программирования, оно остается скрытым в рассмотренных нами примерах – вероятно потому, что они следуют из исходного анализа задачи, который уже исключил состояние в пользу функциональных конструкций. Например, нефункциональная модель оценки стоимости финансового контракта могла бы использовать операцию, преобразующую текущее состояние с обновлением значения вместо функции, строящей последовательность.

Тем не менее мы можем сделать некоторые общие замечания по поводу этого фундаментального решения функциональных подходов. Трудно представить себе какую-либо систему, компьютеризированную или нет, лишенную понятия состояния. Можно даже сказать, что состояние является центральным понятием любых вычислений (приходилось слышать мнение [Peyton Jones, 2007], что программирование без состояния помогает решать проблемы параллелизма, но для общих выводов еще недостаточно фактов). Реальный мир не создает собственных копий в результате любого значимого события. Память наших компьютеров этого также не делает; она просто перезаписывает содержимое своих ячеек. Такие изменения состояния всегда можно смоделировать последовательностью значений, но подобное решение может выглядеть весьма неестественно (как в альтернативном ответе на приведенную ранее загадку: функциональные программисты не заменяют лампочки, а покупают новую розетку, новую электропроводку и новую лампочку).

Современные функциональные языки (в особенности Haskell) признают невозможность игнорирования состояния при таких операциях, как ввод и вывод, а также неуклюжесть более ранних попыток (Peyton Jones and Wadler, 1993). В них появилась концепция *монады* (Wadler, 1995), внедряющей исходные функции в функции более высокого порядка с более сложными сигнатурами; дополнительные компоненты сигнатур могут использоваться для хранения информации состояния, а также любых дополнительных элементов, таких как состояние ошибки (для моделирования обработки исключений) или результатов ввода/вывода.

Использование монад для интеграции состояния обусловлено той же общей идеей (хотя и использованной в противоположном направлении), что и описанная выше методология реализации отложенного поведения посредством моделирования бесконечных последовательностей абстрактными типами данных: чтобы эмулировать в инфраструктуре A возможность T, *неявно* присутствующую в инфраструктуре B, запрограммируйте в A *явную* версию T или ключевого механизма, делающего T возможным. В первом случае T соответствует бесконечным

спискам (а «ключевой механизм» – конечному вычислению бесконечных списков), а во втором – состоянию.

Концепция монады элегантна и, очевидно, полезна для семантических описаний языков программирования (и, особенно, в денотационном семантическом стиле). Однако возникает вопрос, насколько хорошо подходит это решение в качестве механизма, непосредственно используемого программистами. Здесь необходимо очень внимательно проанализировать аргументы. Очевидное возражение против монад – будто рядовых программистов трудно обучить пользоваться ими – несущественно; новаторские идеи, считавшиеся сложными на момент их появления, достаточно легко вливаются в массовое применение, когда преподаватели разрабатывают правильные способы их изложения (и рекурсия, и объектно-ориентированное программирование тоже когда-то считались недоступными для простого «программиста Джо»). Важнее понять, стоит ли игра свеч. Предоставить функциональным программистам доступ к состоянию через монады – все равно, что убедить вашу паству в необходимости целомудрия, а потом сказать, что иметь детей вообще-то неплохо, если они от вас.

Так ли уж необходимо исключать состояние? Пары наблюдений достаточно, чтобы усомниться в этом:

- *Элементарные* операции изменения состояния, такие как присваивание простых значений, имеют четкую математическую модель (логика Хоара, основанная на замене). Тем самым снижается ценность основного преимущества, обычно связываемого с программированием без состояния: упрощение математического обоснования программ.
- Для более *сложных* аспектов установления корректности архитектуры или реализации преимущества функционального подхода не столь очевидны. Например, чтобы доказать, что рекурсивное определение обладает определенными свойствами и имеет завершение, необходим эквивалент инварианта и варианта цикла. Также маловероятно, чтобы эффективные функциональные программы могли позволить себе отказаться от связанных структур данных при всех вытекающих проблемах, нетривиальных независимо от нижележащей модели программирования.

Если функциональное программирование не может значительно упростить задачу установления корректности, остается важный практический аргумент: отсутствие побочных эффектов. Это аналог понятия подстановочности равенства: в математике `f(a)` всегда означает одно и то же для заданных `f` и `a`. Данное свойство истинно для чисто функциональных методологий. В языке программирования, в котором функции

могут иметь побочные эффекты, `f(a)` может возвращать разные результаты при разных вызовах. Устранение такой возможности упрощает понимание текста программы, поскольку мы можем использовать логические стереотипы из математики; например, все привыкли к тому, что `g + g` и `2 × g` означают одно и то же, но если функция `g` имеет побочные эффекты, эквивалентность уже не гарантируется. Сложности возникают не столько для средств автоматизированной проверки (которые могут обнаружить наличие у функции побочных эффектов), сколько у человека, читающего код.

Отсутствие побочных эффектов в выражениях – весьма желательная цель. Однако она не оправдывает исключение понятия состояния из вычислительной модели. Важно вспомнить правило, определенное в методе Eiffel: *принцип разделения команд и запросов* (Meyer, 1997). В этой методологии операции класса четко делятся на две группы: команды, способные изменять целевые объекты, а, следовательно, и состояние, и запросы, выдающие информацию об объекте. Команды не возвращают результата; запросы не могут изменять состояния – иначе говоря, они удовлетворяют правилу отсутствия побочных эффектов. В примере с курсором командами являются `start`, `forth`, и (в конечном случае) `finish`; запросами – `item`, `index`, `count`, `before` и (в конечном случае) `after`. Правило исключает более чем распространенную схему вызова функции для получения результата одновременно с модификацией состояния, которая, вероятно, и является основным источником проблем в императивном программировании. Сначала вы запрашиваете изменение состояния при помощи команды, а затем получаете информацию при помощи запроса (свободного от побочных эффектов). Принцип также можно сформулировать так: «*Заданный вопрос не должен изменять ответа*». Например, из него следует, что типичная операция ввода должна выглядеть так:

```
io.read_character
Result := io.last_character
```

Здесь `read_character` – команда, читающая символ из входного потока, а `last_character` – запрос, возвращающий последний прочитанный символ (обе функции входят в базовую библиотеку ввода/вывода). Непрерывная последовательность вызовов `last_character` гарантированно будет возвращать один и тот же результат. По теоретическим и практическим соображениям, подробно изложенным в другой работе (Meyer, 1997), принцип разделения команд и запросов является методологическим правилом, а не особенностью языка, однако он тщательно соблюдался во всех серьезных программных проектах, написанных на Eiffel, и это принесло значительные преимущества. К сожалению, этот принцип не поддерживается другими школами объектно-ориентированного

программирования (для внесения изменений в них используются вызовы функций в стиле C вместо вызовов процедур), хотя в нашем представлении он является ключевым элементом объектно-ориентированного подхода. На наш взгляд, это реальный способ обеспечения присущего функциональному программированию стремления к отсутствию побочных эффектов – так как выражения, в которых задействованы только запросы, не изменяют состояния, а, следовательно, могут пониматься как в традиционной математике или функциональном языке. При этом понятие команды признает фундаментальную роль концепции состояния для моделирования систем и вычислений.

Объектно-ориентированное представление

А теперь попробуем разобраться, как построить объектно-ориентированную архитектуру для проблем, обсуждавшихся в презентации и статье.

Комбинаторы – хорошо, а типы лучше

До настоящего момента мы имели дело с операциями и комбинаторами. Операции остаются; ключевым шагом должен стать отказ от комбинаторов и замена их типами (или классами – различия проявляются только в области параметризации). Такая замена приводит к значительному повышению уровня абстракции:

- Комбинатор описывает конкретный способ построения нового механизма из уже существующих. Комбинация определяется жестко: скажем, комбинация `взять 3 яблоки` из приводившегося примера связывает один элемент количества с одним элементом еды. Как упоминалось ранее, это математический эквивалент определения структуры ее точной реализацией.
- Класс определяет тип объектов перечислением его возможностей (операций). Он предоставляет абстракцию в смысле абстрактных типов данных: окружающий мир воспринимает объекты исключительно по применяемым к ним операциям, а не по тому, как они конструируются. В соответствии с принципами абстракции данных и объектно-ориентированного проектирования этот подход означает, что объекты известны не по тому, чем они *являются*, а по тому, чем они *обладают* (открытые аспекты и ассоциированные контракты). Данный подход также открывает возможность создания таксономий типов, или *наследования*, для ограничения сложности модели и эффективного использования общности между объектами.

Переходя от первого подхода ко второму, мы ничего не теряем, так как комбинаторы тривиально включаются в классы как особый случай. Достаточно определить возможности с заданием компонентов и соответствующую процедуру создания (конструктор) для объектов. Пример для `take`:

```
class REPETITION create
      make
feature
      base: FOOD
      quantity: REAL
      make (b: FOOD; q: REAL)
        -- Производство элемента из quantity единиц base.
         ensure
             base = b
             quantity = q
         end
 ... Прочие возможности ...
end
```

Для получения объекта этого типа может использоваться запись `create apple_salad.make (6.0, apple)`, эквивалентная выражению с комбинатором.

Программные контракты и параметризация

Так как наше внимание прежде всего направлено на архитектуру, эффект `make` был выражен в форме постусловия, но в действительности с таким же успехом можно было использовать секцию реализации (`do base := b ; quantity := q`). Одним из следствий правильно понимаемого объектно-ориентированного проектирования является сокращение дистанции между спецификацией и реализацией. Мы свободно используем команды, изменяющие состояние, но при этом сохраняем большую часть положительных сторон архитектуры.

В отличие от комбинатора, класс не ограничивается указанными возможностями. Например, он может включать другие процедуры создания – скажем, объединение двух повторений одного компонента:

```
make (r1, r2: REPETITION)
      -- Производство элемента объединением r1 с r2.
      require
          r1.base = r2.base
      ensure
          base = r1.base
          quantity = q
```

Предусловие означает, что смешиваемые компоненты должны относиться к одному базовому типу. Требование также можно преобразовать к статическому виду при помощи системы типов; параметризация (также поддерживаемая в типизованных функциональных языках под странным, хотя и впечатляющим названием «параметрического полиморфизма») приводит нас к следующему определению класса:

```
class REPETITION  [FOOD] create
       ... См. выше ...
feature
      make (r1, r2: REPETITION  [FOOD])
       ... Предусловие не требуется ...
       ... Остальное не изменилось...
end
```

Классы не только могут иметь разные процедуры создания, но обычно обладают гораздо бóльшим количеством методов. А именно, *операции* из предыдущих версий становятся методами соответствующих классов. Классы пудингов (включая классы, описывающие разновидности компонентов, такие как REPETITION) обладают такими методами, как содержание сахара (sugar) и содержание калорий (calorie_content); классы контрактов имеют такие методы, как горизонт (horizon) и стоимость (value). Необходимо сделать пару замечаний:

- Так как мы начинали с чисто функциональной модели, все методы, упоминавшиеся до настоящего момента, являются либо процедурами создания, либо запросами. Хотя функциональный стиль можно сохранить и в объектно-ориентированной инфраструктуре, в ходе разработки в нее могут быть добавлены команды – например, для изменения контракта в ответ на определенное событие (скажем, перезаключение контракта). Впрочем, этот вопрос – быть иль не быть состоянию? – в целом не имеет отношения к теме модульного деления.
- В оригинале функция value порождала бесконечную последовательность. Для сохранения этой сигнатуры можно использовать результат типа COUNTABLE, эквивалент отложенного вычисления; а можно передавать value целочисленный аргумент, чтобы вызов value (i) возвращал i-е значение последовательности.

Политика деления на модули

Модульность, достигнутая до настоящего момента, демонстрирует одну из основополагающих идей объектных технологий (по крайней мере, мы ее считаем основополагающей): *объединение концепций типа и модуля* (Meyer, 1997). В своем простейшем выражении объектно-ориентированный анализ, проектирование и реализация означают, что каждый

модуль системы базируется на некотором типе или объекте, с которым работает система. Эта дисциплина устанавливает более жесткие ограничения, чем модульные средства других методологий: модуль перестает быть простой совокупностью программных элементов – операций, типов, переменных, – которые проектировщик решил держать вместе на основании некоего подходящего критерия. Модуль становится набором свойств и операций, применимых к экземплярам типа.

Класс является результатом этого слияния типов с модулями. В объектно-ориентированных языках, таких как Smalltalk, Eiffel и C# (но не C++ или Java), слияние имеет двустороннюю природу: класс не только определяет тип (или шаблон типа, если задействована параметризация), но и любой тип (включая базовые типы, скажем, целые числа) формально определяется как класс.

Классы также можно ограничить ролью типов, отдельно от модульной структуры. Это особенно справедливо для таких функциональных языков, как OCaml, предоставляющих как традиционную модульную структуру, так и механизм типов, позаимствованный из объектно-ориентированного программирования (а также Haskell с его более ограниченной концепцией класса). И наоборот, возможно снять требование, в соответствии с которым все типы определяются классами, как в C++ и Java, где базовые типы (скажем, целые числа) классами не являются. В нашем представлении объектной технологии происходит полное слияние; при этом необходимо понимать, что высокоуровневая группировка классов (пакеты Java или .NET, кластеры Eiffel) может быть необходимой, но она остается всего лишь организационным средством, а не фундаментальной конструкцией.

Такой подход подразумевает *приоритет типов перед функциями* в том, что касается определения программной архитектуры. Критериями модульного деления являются типы, в которых каждая операция (функция) присоединяется к классу, а не наоборот. Впрочем, функции добиваются компенсации через применение принципов абстрактных типов данных: класс определяется и воспринимается окружающим миром через абстрактный интерфейс (API), перечисляющий возможные операции и их формальные семантические свойства (контракты: предусловия, постусловия и, для класса в целом, – инварианты).

Логическое обоснование такой политики деления на модули заключается в том, что она улучшает модульность системы, в том числе расширяемость, возможность повторного использования и (за счет использования контрактов) надежность. Тем не менее все эти перспективы желательно проанализировать на конкретных примерах.

Наследование

Важнейшим вкладом объектно-ориентированных методов в цели модульности является механизм наследования. Предполагается, что читатель уже с ним знаком, поэтому мы напомним лишь некоторые базовые идеи и очертим их возможные применения в примерах.

Наследование организует классы в таксономии, приблизительно представляющие отношения типа «является частным случаем» – в отличие от другого базового вида отношений между классами – *клиентских* отношений, то есть использования класса через его API (операции, сигнатуры, контракты). В общем случае наследование не обязано соблюдать принцип сокрытия информации, поскольку он несовместим с представлением «является частным случаем». Некоторые авторы ограничивают применение наследования «чистой» субтипизацией, но в действительности нет ничего плохого в его применении для поддержки стандартного механизма включения модулей. В Eiffel используется механизм «неполного наследования» (Ecma International, 2006), который запрещает полиморфизм, но сохраняет все остальные свойства наследования. Двойственная роль наследования вполне соответствует двойственной роли классов как типов и модулей.

В обоих качествах наследование служит для отражения общих качеств. Скажем, для описания элементов нашей вымышленной таксономии пудингов можно использовать граф наследования, показанный на рис. 13.3.

Обратите внимание на распределение ролей между наследованием и клиентскими отношениями. Фруктовый салат (FRUIT_SALAD) является

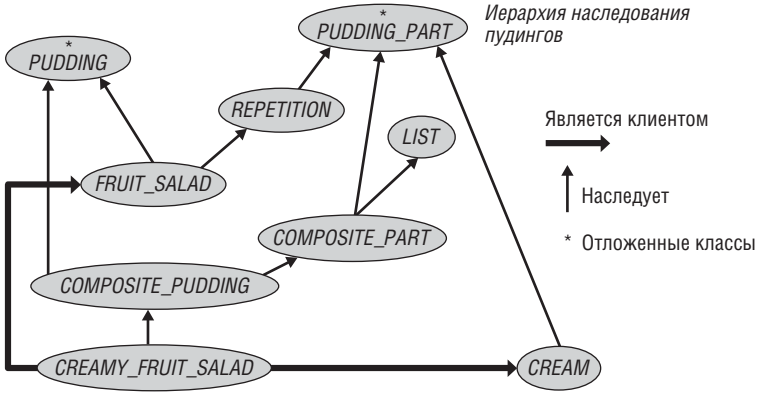


Рис. 13.3. *Диаграмма классов ингредиентов пудинга*

пудингом и повторением (REPETITION) из более раннего примера. Повторение является частным случаем не пудинга (PUDDING), а части пудинга (PUDDING_PART). Некоторые, но не все части пудинга (скажем, составной пудинг COMPOSITE_PUDDING) также являются пудингами. Фруктовый салат является как пудингом, так и повторением (фруктовых частей). С другой стороны, фруктовый салат со сливками (CREAMY_FRUIT_SALAD) не является фруктовым салатом, если понимать под этим термином пудинг, состоящий из одних фруктов. В нем есть и фруктовый салат, и сливки, что изображено на рисунке соответствующими клиентскими связями. Он является составным пудингом, так как это понятие обозначает изделия, которые состоят из нескольких частей (как и более общее понятие COMPOSITE_PART) и при этом одновременно являются пудингами. В данном случае частями, как показывают клиентские связи, являются фруктовый салат и сливки.

Аналогичный подход может быть применен к примеру с контрактами. Типы контрактов делятся на категории: «бескупонные облигации», «опционы» и т. д., полученные на основании тщательного анализа экспертов в этой предметной области.

Множественное наследование абсолютно необходимо для такой объектно-ориентированной формы моделирования. Обратите особое внимание на определение COMPOSITE_PART, в котором применяется стандартный паттерн для описания подобных составных структур (см. Meyer, 1997, 5.1):

```
class COMPOSITE_PART inherit
        PUDDING_PART
        LIST[PUDDING_PART]
feature
  ...
end
```

В квадратных скобках задаются обобщенные параметры. Составная часть одновременно является как частью пудинга со всеми ее свойствами и операциями (содержание сахара и т. д.), так и списком частей пудинга, также со всеми операциями, характерными для списков: перемещениями курсора (start и forth), запросами (item и index), командами вставки и удаления элементов. Элементы списка могут быть пудингами любых из существующих видов, включая, рекурсивно, составные части. Это позволяет применять методы полиморфизма и динамической привязки, которые будут рассмотрены ниже. Обратите внимание на полезность совмещения параметризации и наследования. Также множественное наследование должно быть представлено полноценным механизмом классов, а не ограниченной интерфейсной формой (как в Java и .NET), которая для данного случая не подходит.

Полиморфизм, полиморфные контейнеры и динамическая привязка

Вклад наследования и параметризации в расширяемость частично представлен методами полиморфизма и динамической привязки, как показывает следующая версия `sugar_content` класса `COMPOSITE_PART` (рис. 13.4):

```
sugar_content: REAL
      do
          from start until after loop
               Result := Result + item.sugar_content
               forth
          end
      end
```

Операции класса `LIST` применяются непосредственно к производному от него классу `COMPOSITE_PART`. Результат `item` может относиться к любому из типов, производных от `PUDDING`; так как в дальнейшем переменная может использоваться для обозначения объектов разных типов, она называется полиморфной переменной (в данном случае точнее назвать ее *полиморфным запросом*). Вся структура `COMPOSITE_PART`, содержащая элементы разных типов, называется *полиморфным контейнером*. Полиморфные контейнеры возможны благодаря комбинации полиморфизма, который сам по себе является результатом наследования и параметризации (так как речь идет о двух совершенно разных механизмах, обозначение параметризации принятым в функциональном программировании термином «параметрический полиморфизм» приведет к путанице).

Полиморфизм `item` подразумевает, что последовательные вызовы `item.sugar_content` будут применяться к объектам разных типов; в соответствующих классах могут быть определены разные версии запроса `sugar_content`. *Динамическая привязка* гарантирует, что такие вызовы будут в каждом случае применены к правильной версии в зависимости

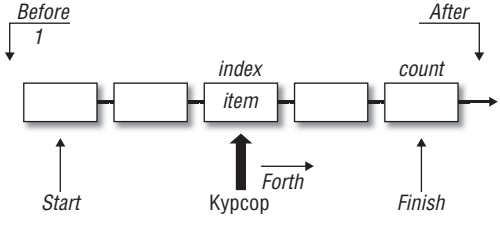


***Рис. 13.4.** Полиморфный список с курсорами*

от фактического типа объекта. Если часть сама по себе является составной, это будет приведенная выше версия, примененная рекурсивно, но также может быть и любая другая – например, версия для CREAM.

Как и в большинстве современных подходов к объектно-ориентированному проектированию, полиморфизм работает под управлением системы типов. Значения элемента относятся к переменному типу, но только среди потомков PUDDING, как указывает параметр COMPOSITE_PART. Этот аспект разработки повлиял как на мир функционального программирования, так и на объектно-ориентированный мир: применение все более усложняющихся систем типов (по-прежнему основанных на небольшом числе простых концепций, таких как наследование, полиморфизм и параметризация) для отражения все большего объема сведений об архитектуре системы в ее структуре типов.

Отложенные классы и методы

На приведенной ранее структурной диаграмме классы PUDDING и PUDDING_PART помечены как «отложенные» (deferred) (звездочка в синтаксисе объектно-ориентированного моделирования BON [Walden and Nerson, 1994]). Это означает, что они не имеют полностью определенной реализации; также используется термин «абстрактный класс». Отложенный класс обычно обладает отложенными методами, т. е. у него имеется сигнатура и (что еще важнее) контракт, но нет реализации. Реализации появляются в неотложенных («конкретных») классах-потомках, адаптированные к тем решениям, которые принимаются каждым конкретным классом для реализации общих концепций, определяемых абстрактным классом. В нашем примере оба класса PUDDING и PUDDING_PART обладают отложенными методами sugar_content и calories; потомки конкретизируют их – например, в COMPOSITE_PART содержание сахара определяется суммированием содержаний всех частей, как показано выше. В COMPOSITE_PUDDING, наследующем эту версию от COMPOSITE_PART и отложенную версию от PUDDING, реализация определяется конкретной версией, обладающей более высоким приоритетом.

Примечание

При наследовании двух методов с одинаковыми именами возникает конфликт имен, который должен быть разрешен посредством переименования. Исключением является ситуация, в которой один из методов является отложенным, а другой – конкретным. В этом случае формируется один метод с имеющейся реализацией. Именно в таких применениях механизма наследования перегрузка имен создает массу хронических сложностей, и возникает впечатление, что этот механизм не должен присутствовать в объектно-ориентированных языках.

Отложенные классы сложнее концепции «интерфейсов» Java и .NET. Во-первых, они могут связываться с контрактами, ограничивающими будущую конкретизацию, а, во-вторых, в них могут присутствовать конкретные методы, вследствие чего отложенные классы заполняют весь спектр между полностью отложенным классом, ограничивающимся чистым описанием реализации, и конкретным классом, полностью определяющим реализацию. Возможность описания частичных реализаций исключительно важна для объектно-ориентированного проектирования и архитектуры.

В примере с финансовыми контрактами `CONTRACT` и `OPTION` будут естественными кандидатами для оформления в виде отложенных классов, хотя они и не обязаны быть *полностью* отложенными.

Оценка и улучшение модульности в объектно-ориентированных архитектурах

В предыдущем разделе представлен краткий обзор применения объектно-ориентированных архитектурных методов в наших примерах. Теперь необходимо проанализировать схематичный результат в свете критериев модульности, представленных в начале обсуждения. Надежность в основном зависит от системы типов и контрактов; наше внимание будет сосредоточено на возможностях повторного использования и расширения.

Повторное использование операций

Одно из важнейших последствий применения наследования заключается в возможности перемещения общих методов на самый верхний возможный уровень абстракции. Далее потомкам не обязательно повторять реализацию унаследованных методов: они просто наследуют их «как есть». Если им понадобится изменить реализацию с сохранением функциональности, они просто переопределяют унаследованную версию. Под «сохранением функциональности» в данном случае понимается соблюдение исходных контрактов независимо от того, конкретизирована переопределяемая версия или все еще остается отложенной. Такой подход хорошо сочетается с динамической привязкой: клиент может использовать операцию на верхнем уровне абстракции (например, `my_pudding.sugar_content` или `my_contract.value`), не зная, какая версия операции используется, в каком классе и была ли она определена в этом классе или унаследована им.

Благодаря наследованию общих аспектов количество определений может оказаться существенно ниже максимального значения `t×f`. Любое

сокращение, безусловно, полезно. Общее правило проектирования программных продуктов гласит, что любые повторения всегда потенциально вредны, поскольку они становятся источником будущих трудностей с управлением конфигурацией, сопровождением и отладкой (если в оригинале будет допущена ошибка, ее также придется исправлять в каждой из копий). Как справедливо заметил Дэвид Парнас, копирование/вставка – враг программиста.

Степень реального сокращения очевидным образом зависит от качества структуры наследования. Здесь следует руководствоваться принципами абстрактных типов данных: поскольку ключом к определению типов при объектно-ориентированном проектировании является анализ применимых операций, в правильно спроектированной иерархии наследования на верхний уровень будут выведены классы, содержащие общие аспекты многих разновидностей.

У этой методологии не существует четкого эквивалента в функциональной модели. При использовании комбинаторов необходимо определять разновидность каждой операции для каждой комбинации с повторением всех общих операций.

Расширяемость: добавление типов

Насколько хорошо объектно-ориентированная форма архитектуры поддерживает расширяемость? Одним из самых распространенных видов расширения системы является добавление новых типов: новой разновидности пудинга, новой части пудинга, нового финансового контракта. В подобных ситуациях объектная технология проявляется во всей красе. Просто найдите в структуре наследования место, к которому новая разновидность лучше всего подходит (в смысле наличия большинства общих операций) и напишите новый класс, который наследует некоторые методы, переопределяет или изменяет те, для которых он имеет собственные разновидности, и добавляет все новые методы и инварианты, соответствующие новой логической сущности.

Динамическая привязка снова играет важную роль; преимущество объектно-ориентированного подхода заключается в том, что он избавляет клиентские классы от необходимости выполнять многоуровневое ветвление для выполнения операций: «если это фруктовый салат, то обработать его так, иначе, если это флан, то обработать его этак, иначе...». Такие проверки должны повторяться для каждой операции и, что еще хуже, – должны обновляться в каждом клиенте и в каждой операции при каждом добавлении или изменении типа. Такие структуры, требующие, чтобы клиентские классы обладали полной информацией о структуре концепций, от которых они зависят, становились главным

источником деградации и старения архитектур до появления объектно-ориентированных методологий. Динамическая привязка решает проблему; клиентское приложение запрашивает `my_contract.value`, а внутренняя реализация сама выбирает подходящую версию.

Ни одна другая методология программных архитектур не сравнится с изяществом этого решения, объединяющего самые сильные стороны объектно-ориентированных методологий.

Расширяемость: добавление операций

Оценки удобства расширения объектных архитектур отчасти (помимо таких механизмов, как сокрытие информации и параметризация, а также центральная роль контрактов) базируются на предположении о том, что самые значительные изменения в жизни системы относятся к только что рассмотренной разновидности: введению типов, которые используют некоторые операции существующих типов, но также могут определять новые операции. В самом деле, практический опыт говорит о том, что именно они являются самым частым источником нетривиальных изменений в реальных системах, в которых проявляются преимущества объектно-ориентированных методов перед другими. Но как насчет другого случая: добавления операций в существующие типы? Допустим, некоторое клиентское приложение, использующее концепцию пудинга, может захотеть вычислить стоимость изготовления различных пудингов, хотя сами классы пудингов не имеют соответствующего метода.

Функциональное программирование справляется с добавлением операций не лучше и не хуже, чем с добавлением типов: все сводится к увеличению на 1 значения `f` вместо `t`. Однако для объектно-ориентированных решений такая нейтральность не характерна. Простейшее решение заключается в добавлении новой возможности на правильно выбранном уровне иерархии. Однако при этом возможны два потенциальных недостатка:

- Так как наследование создает относительно сильную привязку («является частным случаем») между классами, изменение распространяется на всех существующих потомков. В общем случае добавление нового метода в класс, находящийся на высоком уровне иерархической структуры, может оказаться делом весьма нетривиальным.
- Такое решение попросту невозможно, если автору клиентской системы не разрешено изменять исходные классы или он не имеет доступа к их тексту. Такая ситуация часто возникает на практике, особенно если классы сгруппированы в библиотеки (скажем, биб-

лиотеку финансовых контрактов). Неразумно разрешать изменять библиотеку автору каждого приложения, в котором она используется.

Основных объектно-ориентированных методов (Meyer, 1997) здесь недостаточно. Стандартное и часто применяемое на практике объектно-ориентированное решение основано на паттерне «*Посетитель*» (Gamma et al., 1994). Следующая схема, хотя и не совсем точно совпадает со стандартным представлением, дает общее представление об идее (приводится по материалам книги Мейера «Touch of Class: An Introduction to Programming Well» (2008), вводного учебника по программированию для первокурсников – отсюда видно, насколько фундаментальными стали эти концепции). На рис. 13.5 изображены основные участники паттерна.

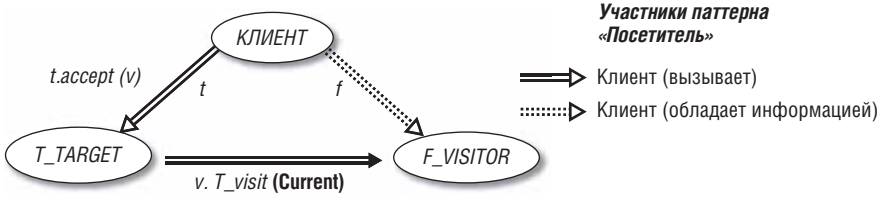


***Рис. 13.5.** Акторы паттерна «Посетитель»*

Паттерн превращает дуэт приложения (класс `CLIENT`) и существующих типов (`T_TARGET` для конкретного типа `T`, которым может быть `PUDDING` или `CONTRACT` в наших примерах) в трио; для этого вводится класс-посетитель `F_VISITOR` для каждой новой операции `F` (например, `COST_VISITOR`). Класс приложения (`CLIENT`) вызывает операцию для целевого объекта, передавая посетителя в аргументе:

```
my_fruit_salad.accept (cost_visitor)
```

Команда `accept (v: VISITOR)` выполняет операцию, вызывая для своего *аргумента* `v` (`cost_visitor` в данном примере) функцию `FRUIT_SALAD_visit`, имя которой определяет тип целевого объекта. Функция является частью класса, описывающего целевой класс, в данном случае `FRUIT_SALAD`; она применяется к объекту соответствующего типа, который передается в аргументе `T_visit`. `Current` на рисунке – принятое в Eiffel обозначение текущего объекта (аналог `this` или `self` в других языках). *Целевой класс* вызова (`v` на рисунке) определяет операцию, используя объект соответствующего типа посетителя (например, `COST_VISITOR`).

Ключевым вопросом при оценке расширяемости в программных архитектурах всегда является распределение информации. Метод может

достигнуть расширяемости только за счет ограничения количества информации, которой модули должны располагать друг о друге (чтобы добавление или изменение модуля имело минимальные последствия для существующих структур). Чтобы понять сложную хореографию паттерна «Посетитель», полезно посмотреть, что должен или не должен знать каждый участник:

- Целевой класс знает конкретный тип, а также его контекст в иерархии типов (так как, например, `FRUIT_SALAD` наследует от `COMPOSITE_PUDDING`, а `COMPOSITE_PUDDING` – от `PUDDING`). Он *не знает* о новых операциях, вызываемых извне, таких как вычисление стоимости изготовления пудинга.
- Класс-посетитель знает все о конкретной операции (вычисление стоимости) и предоставляет ее разновидности для диапазона типов, обозначая объекты при помощи аргументов: именно здесь обнаруживаются такие функции, как `fruit_salad_cost`, `flan_cost`, `tart_cost` и т. д.
- Класс-клиент должен применять заданную операцию к объектам указанных типов, поэтому он должен знать эти типы (только факт их существования, но не другие свойства) и операции (только факт их существования и применимость к заданным типам, но не конкретные алгоритмы в каждом случае).

Некоторые из необходимых операций, такие как `accept` и `T_visit`, предоставляются предками. На рис. 13.6 изображена общая диаграмма наследования (имя класса `FRUIT_SALAD` сокращено до `SALAD`).

Такая архитектура обычно используется для включения новых операций в существующей структуре со многими вариантами наследования без необходимости изменять структуру для каждой операции. Стандартный пример применения встречается в области обработки языков (компиляторы и другие инструменты IDE), где нижележащей структурой является абстрактное синтаксическое дерево AST (Abstract Syntax Tree): необходимость обновлять класс AST каждый раз, когда новому инструменту потребуется для его собственных целей выполнить операцию перебора узлов, с применением к каждому узлу операции по желанию этого инструмента (это называется «посещением» узлов, что объясняет название паттерна «Посетитель»).

Чтобы эта схема работала, клиенты должны иметь возможность выполнить `t.accept (v)` для любого `t` любого целевого типа. Отсюда следует, что все целевые типы наследуют от общего класса (`PUDDING` в нашем примере), в котором `accept` объявляется в отложенной форме. Это довольно щекотливое требование, потому что целью всего построения было именно предотвращение модификации существующих целевых классов.

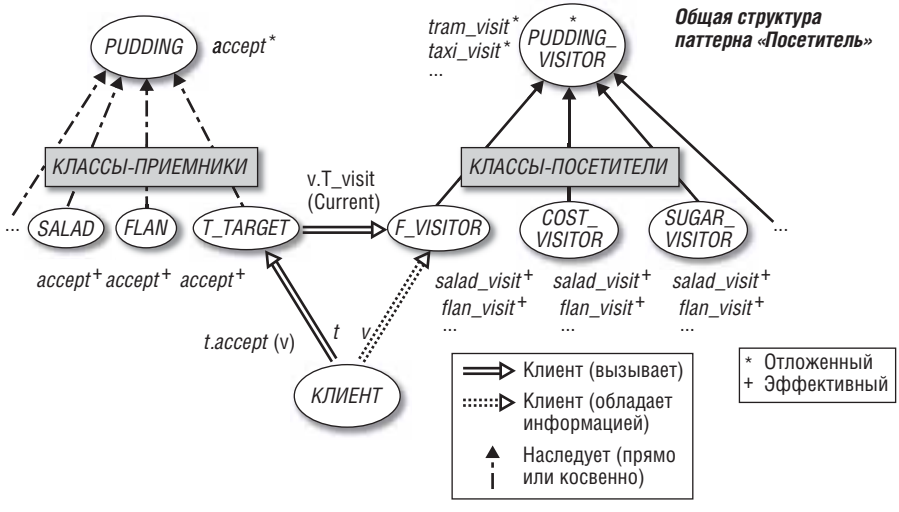


Рис. 13.6. *Общая диаграмма: архитектура для конструирования пудингов*

Проектировщики, использующие паттерн «Посетитель», обычно считают это требование приемлемым, поскольку для его выполнения достаточно проследить за тем, чтобы все задействованные классы имели общего предка (довольно часто это требование выполняется изначально, если они представляют собой разновидности общей концепции – такой как `PUDDING` или `CONTRACT`), и добавить в предка всего *один* отложенный метод `accept`.

Паттерн «Посетитель» широко применяется на практике. Пусть читатель сам судит о том, насколько он «красив». На наш взгляд, бывают и покрасивее. Основные претензии:

- Необходимость общего предка со специальным методом `accept` в классах предметной области, которые не должны отягощаться концепциями, не относящимися к области применения (будь то пудинги, финансовые контракты или что-нибудь еще).
- Еще большее беспокойство вызывает размножение классов с многочисленными миниатюрными классами `F_VISITOR`, воплощающими крайне специфическую информацию (специальная операция над набором специальных типов). В контексте общей программной архитектуры это воспринимается как загрязнение.

Чтобы избежать загрязнения, необходимо включить в базовую объектно-ориентированную инфраструктуру принципиально новую концепцию: агентов.

Агенты: упаковка операций в объектах

Основная идея агентов (включенных в базовую объектно-ориентированную инфраструктуру Eiffel в 1997 году; в C# используется термин «делегаты») выражается словами, напоминающими литературу по функциональному программированию: операции (функции в функциональном программировании, методы в объектно-ориентированном программировании) рассматриваются как «полноценные сущности». В контексте ОО полноценными сущностями во время выполнения являются только объекты, по своей статической структуре соответствующие классам.

Механизм агентов

Агентом называется объект, представляющий метод некоторого класса, готовый к вызову. Вызов метода `x.f(u, ...)` полностью определяется именем метода `f`, целевым объектом `x` и аргументами `u, ...`. Агентское выражение, в котором указывается `f` и могут указываться целевой объект и аргументы (нуль, некоторые или все из них), называется *закрытым*. Все остальные выражения, не указанные в определении агента, называются *открытыми*. Выражение обозначает объект, представляющий метод с закрытыми аргументами, которыми присвоены заданные значения. Одной из операций, которые могут выполняться с агентом, является операция `call`, представляющая вызов `f`; если у агента имеются открытые аргументы, соответствующие значения должны быть переданы в аргументах `call` (для закрытых аргументов используются значения, уже заданные в определении агента).

Простейший пример агентского выражения – `agent f`. В данном случае все аргументы являются открытыми, но целевой объект закрыт. Следовательно, если `a` является агентским выражением (в результате присваивания `a := agent f` или вызова `p (agent f)` с формальным аргументом `a`), то выражение `a.call ([u, v])` приводит к тому же эффекту, что и `f (u, v)`. Конечно, различие заключается в том, что `f (u, v)` прямо задает имя метода (хотя динамическая привязка означает, что он может быть одной из разновидностей известного метода), тогда как в форме с агентами `a` – всего лишь имя, которое может быть получено из другого программного модуля. Таким образом, в этой точке программы о методе неизвестно ничего, кроме его сигнатуры (и, возможно, контракта). Так как `call` является библиотечной функцией общего назначения, ей должен передаваться один аргумент. Проблема решается использованием *кортежа* (*tuple*), в данном случае двухэлементного кортежа `[u, v]`. В такой форме, `agent f`, целевой объект закрыт (им является текущий объект), а оба аргумента открыты.

Другая разновидность – `agent x.f`. Здесь аргументы тоже открыты, а целевой объект закрыт: на этот раз им является объект `x` вместо текущего объекта. Чтобы сделать целевой объект открытым, используется запись `agent {T}.f`, где `T` относится к типу `x`. В этом случае `call` понадобится кортеж из трех аргументов: `a.call ([x, u, v])`. Чтобы оставить некоторые аргументы открытыми, можно использовать аналогичную запись вида `agent x.f ({U}, v)` (типичный вызов имеет вид `a.call ([u])`), но, поскольку тип `U` объекта `u` очевиден из контекста, указывать его явно не обязательно; достаточно поставить вопросительный знак, как в выражении `agent x.f (?, v)`. Отсюда также следует, что исходные формы со всеми открытыми аргументами, `agent f` и `agent x.f`, являются сокращениями для `agent f (?, ?)` и `agent x.f (?, ?)`.

Механизм `call` использует динамическую привязку; используемая версия `f`, как и при «неагентском» вызове, зависит от динамического типа целевого объекта.

Если `f` представляет запрос, а не команду, для получения результата вызова `f` от соответствующего агента можно использовать `item` вместо `call`, например `a.item ([x, u, v])` (вызов с возвращением значения результата); также можно использовать `call`, а затем обратиться к `a.last_result`, которое согласно принципу разделения команд и запросов вернет то же значение без дополнительных вызовов `call` при последующих вызовах.

Возможно и нетривиальное использование агентов: вместо того чтобы определять агента на базе существующего метода `f`, можно записать его во встроенном коде, как в выражении `editor_window.set_mouse_enter_action (agent do text.highlight end)`, демонстрирующем типичное применение агентов в графических интерфейсах; это базовый стиль событийного программирования в библиотеке EiffelVision (Eiffel Software: документация EiffelVision). Механизм встроенной записи агентов аналогичен лямбда-выражениям в функциональных языках: записанные вами операции становятся непосредственно доступными для программного кода как значения, с которыми можно работать как с любыми другими «полноценными сущностями».

В более обобщенной формулировке агенты позволяют объектно-ориентированным инфраструктурам определять высокоуровневые функциональные объекты – такие же как в функциональных языках и обладающие такими же выразительными возможностями.

Область применения агентов

Агенты оказались важным и естественным компонентом базовых объектно-ориентированных механизмов. В частности, они широко используются в следующих целях:

- Итеративный перебор: применение переменной операции, которая естественным образом представляется агентом, ко всем элементам контейнерной структуры.
- Программирование GUI (см. выше).
- Математические вычисления – например, интегрирование некоторой функции, представленной агентом, по некоторому интервалу.
- Рефлексия, когда агенты предоставляют свойства методов (не только возможность вызывать их через `call` и `item`) и классов.

Агенты сыграли важную роль в нашем исследовании того, как заменить паттерны проектирования компонентами многократного использования (Arnout 2004; Arnout and Meyer, 2006; Meyer, 2004; Meyer and Arnout, 2006). Идея в том, что если проектировщику приложения понадобится паттерн, то ему придется изучать этот паттерн во всех подробностях (включая архитектуру и реализацию) и строить «с нуля» в своем приложении, тогда как компонент может просто использоваться через API. Среди успешных примеров такого рода можно назвать паттерн проектирования «Наблюдатель» (Meyer, 2004; Meyer, 2008); никому, кто видел решение на базе агентов, уже не захочется снова пользоваться этим паттерном. Другие примеры – паттерны «Фабрика» (Arnout and Meyer, 2006) и «Посетитель» – будут рассматриваться ниже.

Агентская библиотека для замены паттерна «Посетитель»

Механизм агентов предоставляет гораздо более удачное решение проблемы, которая несколько неуклюже решается паттерном «Посетитель», а именно –добавление операций к существующим типам без изменения исходных классов. Решение подробно описано у Мейера и Арну (2006) и доступно в виде библиотеки с открытым кодом на сайте Цюрихского федерального технологического института *http://se.ethz.ch*.

Итоговый клиентский интерфейс отличается исключительной простотой. В целевые классы (`PUDDING`, `CONTRACT` и т. д.) не нужно вносить никакие изменения; метод `accept` более не требуется. Классы можно использовать непосредственно в их текущем виде; больше нет размножения классов-посетителей, есть только единый библиотечный класс `VISITOR` всего с двумя методами для основного использования, `register` и `visit`.

Проектировщику клиента не нужно разбираться во внутреннем устройстве этого класса или беспокоиться о реализации паттерна «Посетитель». Достаточно применить базовую схему использования API:

1. Объявить переменную, представляющую объект-посетитель, указать высокоуровневый тип цели при помощи параметра `VISITOR` и создать соответствующий объект:

```
pudding_visitor: VISITOR [PUDDING]
create pudding_visitor
```

2. Для каждой операции, выполняемой с объектами определенного типа в целевой структуре, зарегистрировать соответствующего агента у посетителя:

```
pudding_visitor.register (agent fruit_salad_cost)
```

3. Чтобы выполнить операцию с определенным объектом (обычно в процессе перебора), используйте метод `visit` библиотечного класса `VISITOR`:

```
pudding_visitor.visit (my_pudding)
```

Вот и весь интерфейс: один объект-посетитель, регистрация применимых операций и единственная операция `visit`. Эта простота объясняется тремя свойствами.

- Запись применяемых операций (таких как `fruit_salad_cost`) не должна зависеть от выбора архитектуры. Часто они будут доступны в виде функций, что сделает возможной запись `agent fruit_salad_cost`; если нет (особенно если это очень простые операции), клиент может воспользоваться встроенным кодом агента. В любом случае необходимость в лишних методах `T_visit` отпадает.
- На первый взгляд немного странно, что для добавления посетителя достаточно всего одного класса `VISITOR` с одной функцией `register`. В решении, основанном на паттерне «Посетитель», вызов `t.accept (v)` и `t` определяли тип цели (конкретный вид пудинга), но `register` такую информацию не указывает. Как механизм находит правильную разновидность применяемой операции (стоимость фруктового салата или стоимость флана)? Ответ на этот вопрос является следствием рефлексивных свойств механизма агентов: объект агента воплощает всю информацию об ассоциированном методе, включая его сигнатуру. Таким образом, `agent fruit_salad_cost` включает информацию о том, что эта функция применима к фруктовым салатам (из сигнатуры `fruit_salad_cost (fs: FRUIT_SALAD)`, также доступной в случае встроенного агента из его текста). Это позволяет организовать внутренние структуры данных `VISITOR` таким образом, что при вызо-

ве вида `pudding_visitor.visit (my_pudding)` функция `visit` найдет правильную функцию (или функции) на основании динамического типа цели, в данном случае `pudding_visitor: VISITOR [P]` для конкретного типа пудинга `P`, также соответствующую типу объекта, динамически связанному с аргументом, в данном случае полиморфным `my_pudding` (за этим на статическом уровне проследит система типов).

- Методология также пользуется преимуществами наследования и динамической привязки: если функция зарегистрирована для обобщенного типа пудинга (допустим, `COMPOSITE_PUDDING`) и нет функции, зарегистрированной для более конкретного типа (например, если стоимость всех составных пудингов может вычисляться одним способом), `visit` использует самое близкое совпадение.

В том виде, в котором механизм описан, он дополняет традиционный объектно-ориентированный механизм. Когда проблема заключается в добавлении типов, предоставляющих разновидности существующих операций, наследование и динамическая привязка творят чудеса. Для проблемы добавления операций в существующие типы без изменения этих типов подойдет описанное решение.

Применяя упоминавшийся выше критерий оценки модульности по распределению информации (кто и что должен знать?), мы видим, что в этом подходе:

- Целевые классы знают только о фундаментальных операциях (таких как `sugar_content`), характеризующих соответствующие типы.
- Приложению необходимо знать только интерфейс используемых им целевых классов, а также два важнейших метода, `register` и `visit`, библиотечного класса `VISITOR`. Если ему понадобятся новые операции с целевыми типами, не учтенные при проектировании целевых классов (такие как `cost` из нашего примера), ему необходимо только предоставить разновидности операций для целевых типов, представляющих интерес. При этом необходимо понимать, что в отсутствии переопределяющей регистрации для конкретных типов будут использоваться обобщенные операции.
- Библиотечный класс `VISITOR` ничего не знает ни о типах целей, ни о приложениях.

Кажется, сократить необходимую информацию для разных частей системы уже невозможно. По нашему мнению, остается всего один открытый вопрос: должен ли столь фундаментальный механизм оставаться доступным на уровне библиотеки или для него следует определить специальную языковую конструкцию?

Оценка

При первом появлении агентов возникли опасения, что они могут создать избыточность, а, следовательно, и путаницу, предлагая альтернативные решения в ситуациях, решаемых стандартными средствами ОО (эти опасения особенно сильны в Eiffel, проектирование которого велось по принципу предоставления «одного хорошего способа для решения любой задачи»). Этого не произошло: агенты заняли свое законное место в объектно-ориентированном арсенале, а проектировщики без особого труда определяют, когда они уместны, а когда нет.

На практике любые нетривиальные применения агентов (в частности, уже упоминавшиеся замены паттернов) также зависят от параметризации, наследования, полиморфизма, динамической привязки и других нетривиальных объектно-ориентированных механизмов. Все это укрепляет наше убеждение в том, что данный механизм является необходимым компонентом успешных объектных технологий.

Примечание

Противоположное мнение приводится в технической статье Sun, объясняющей, почему в Java не нужны аналоги агентов или делегатов (Sun Microsystems, 1997). В статье показано, как этот механизм эмулируется при помощи «внутренних классов» Java. Статья интересная и хорошо аргументированная, но, на наш взгляд, ей в основном удается доказать прямо противоположный тезис. Внутренние классы действительно справляются с этой задачей, но, как и при устранении паттерна «Посетитель» с его размножением крошечных классов, приведенное решение с использованием внутренних классов наглядно демонстрирует простоту, элегантность и модульность, присущие решению с использованием агентов.

Агенты, как упоминалось ранее, позволяют объектно-ориентированным инфраструктурам предоставлять те же выразительные возможности, как в функциональном программировании, при помощи обобщенного механизма определения высокоуровневых функциональных объектов (операций, которые могут использовать операции, рекурсивно проявляющие то же свойство, в качестве входных и выходных данных). Даже у лямбда-выражений имеются свои аналоги в виде встроенных агентов. Эти механизмы формировались под очевидным влиянием функционального программирования и теоретически должны вызвать энтузиазм у его сторонников, хотя существуют опасения, что некоторые разработчики будут считать их данью уважения, которую порок платит добродетели (La Rochefoucauld, 1665). Если не обращать внимания на синтаксис, единственное принципиальное различие заключается в том, что агенты могут инкапсулировать не только чистые

функции (запросы), но и команды. Тем не менее обеспечение полной чистоты (отсутствия побочных эффектов) не имеет особого отношения к обсуждениям архитектуры, по крайней мере при соблюдении принципа разделения команд и запросов, сохраняющего главную практическую ценность – отсутствие побочных эффектов у выражений – без насильственного переведения модели с состоянием в искусственный мир моделей без состояния.

Возможно, агенты являются «последним штрихом» в картине вклада объектных технологий в модульность, но это всего лишь один из ее элементов – как кратко представленных в дискуссии, так и тех, которые в ней не упоминались. Именно комбинация этих элементов, выходящая за рамки того, что может предложить функциональный подход, делает объектно-ориентированное проектирование лучшей из имеющихся методологий для построения красивых архитектур.

Благодарности

Я благодарен всем, кто высказал свои важные замечания по поводу чернового варианта этой статьи; как известно, благодарность еще не означает одобрения (это особенно очевидно в случае гуру функционального программирования, которые любезно поделились своим конструктивным мнением, притом что мои аргументы, как я подозреваю, их нисколько не поколебали). Особенно полезными были замечания Саймона Пейтона Джонса (Simon Peyton Jones), Эрика Мейера (Erik Meijer) и Диомидиса Спинеллиса (Diomidis Spinellis). Ответы Джона Хьюза на мои вопросы по поводу его классической статьи были подробными и понятными. Библиотека реализации паттерна «Посетитель», описанная в завершающей части главы, создана Карин Арну (Karine Arnout) (ныне Карин Безо, Karine Bezault). Я благодарен Глории Мюллер (Gloria Müller) за любопытные наблюдения, которые я почерпнул из ее магистерской диссертации ETH по реализации библиотеки Haskell-подобной функциональности в Eiffel. Я особенно благодарен редакторам этой книги, Диомидису Спинеллису и Георгиосу Гусиосу, за возможность опубликовать эту статью, а также за их исключительное терпение, с которым они относились к задержкам со сдачей материала.

Библиография

Arnout, Karine. 2004. «From patterns to components». Ph.D. thesis, ETH Zurich. *http://se.inf.ethz.ch/people/arnout/patterns/*.

Arnout, Karine, and Bertrand Meyer. 2006. «Pattern componentization: the Factory example». Innovations in Systems and Software Technology

(a NASA Journal). New York, NY: Springer-Verlag. *http://www.springerlink.com/content/am08351v30460827/*.

Eber, Jean-Marc, на основании совместной теоретической работы с Саймоном Пейтоном Джонсом (Simon Peyton Jones) и Пьером Вайсом (Pierre Weis). 2001. «Compositional description, baluation, and management of financial contracts: the MLFi language». *http://www.lexifi.com/Downloads/MLFiPresentation.ppt*.

Ecma International. 2006 «*Eiffel: Analysis, Design and Programming Language*». ECMA-367. *http://www.ecma-international.org/publications/standards/Ecma-367.htm*.

Frankau, Simon, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. 2008. «Commercial uses: Going functional on exotic trades». *Journal of Functional Programming*, 19(1):2745, October.

Gamma, Erich, et al. 1994. «*Design Patterns: Elements of Reusable Object-Oriented Software*». Boston, MA: Addison-Wesley.[1]

Hughes, John. 1989. «Why functional programming matters.» *Computer Journal*, vol. 32, no. 2: 98–107 (revision of a 1984 paper). *http://www.cs.chalmers.se/?rjmh/Papers/whyfp.pdf*.

La Rochefoucauld, Franзois de. 1665. *Réflexions ou sentences et maximes morales*.

Meyer, Bertrand, and Karine Arnout. 2006. «Componentization: the Visitor example». *Computer* (IEEE), vol. 39, no. 7: 23–30. *http://se.ethz.ch/?meyer/publications/computer/visitor.pdf*.

Meyer, Bertrand. 1992. «*Eiffel: The Language*» (Second Printing). Upper Saddle River, NJ: Prentice Hall.

Meyer, Bertrand. 1997. «*Object-Oriented Software Construction*», Second Edition. Upper Saddle River, NJ: Prentice Hall. *http://archive.eiffel.com /doc/oosc/*.

Meyer, Bertrand. 2004. «The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design». *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, *Lecture Notes in Computer Science* 2635, pp. 236–271. New York, NY: Springer-Verlag. *http://se.ethz.ch/?meyer/publications/lncs/events.pdf*.

Meyer, Bertrand. 2008. «*Touch of Class: An Introduction to Programming Well*». New York, NY: Springer-Verlag. *http://touch.ethz.ch*.

[1] Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001–2007.

Peyton Jones, Simon, Jean-Marc Eber, and Julian Seward. 2000. «Composing contracts: An adventure in financial engineering». Functional pearl, in *ACM SIGPLAN International Conference on Functional Programming* (ICFP '00), Montreal, Canada, September '00. ACM Press, pp. 280–292. *http://citeseer.ist.psu.edu/jones00composing.html*.

Peyton Jones, Simon, and Philip Wadler. 1993. «Imperative functional programming». *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, pp. 71–84. *http://citeseer.ist.psu.edu/peytonjones93imperative.html*.

Steingold, Sam. Online at *http://www.podval.org/?sds/ocaml-sucks.html*.

Sun Microsystems. 1997. «About Microsoft's 'Delegates'». White paper by the Java Language Team at JavaSoft. *http://java.sun.com/docs/white/delegates.html*.

Wadler, Philip. 1995. «Monads for functional programming». *Advanced Functional Programming*, Lecture Notes in Computer Science 925. Eds. J. Jeuring and E. Meijer. New York, NY: Springer-Verlag. *http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf*.

Walden, Kim, and Jean-Marc Nerson. 1994. «*Seamless Object-Oriented Software Architecture*». Upper Saddle River, NJ: Prentice Hall. *http://www.bon-method .com/index_normal.htm*.

14

Перечитывая классику

Панайотис Луридас

Похоже, в любой научной области существуют выполненные работы и люди, без упоминания которых никак не обойтись. Вероятно, чемпионом современности в этом отношении является Ноам Хомски. Согласно статье из MIT Tech Talk за апрель 1992 года Хомски был самым упоминаемым ученым за последние 20 лет. По данным Индекса цитирования в области культуры и искусства, в «первую десятку» входят Маркс, Ленин, Шекспир, Аристотель, Библия, Платон, Фрейд, Хомски, Гегель и Цицерон. Индекс цитирования в области науки утверждает, что за период с 1972 по 1992 год Хомски упоминался 1619 раз.

В области программирования лавровый венок, вероятно, достанется книге «Design Patterns: Elements of Reusable Object-Oriented Software»[1] (также известна под названием книги «Банды четырех», 1994). Поиск в Google по точному названию книги возвращает около 173 000 результатов (по состоянию на весну 2008 года). Если обратиться к более академическому контексту, поиск в библиотеке ACM Digital Library

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2007.

возвращает 1572 результата. Разработчики паттернов проектирования образовали одно из самых ярких и энергичных сообществ в области программирования за последние 20 лет. Трудно представить себе современного программиста, который бы не был знаком с этой важной книгой.

Автор данной главы охотно воздает должное этим источникам, увеличивая каждый из их счетчиков упоминаний еще на единицу.

Заслуга книги «Банды четырех» заключалась в том, что она донесла паттерны проектирования до широких масс. Кроме того, для движения паттернов она стала не только отправной, но и конечной точкой: материалы, относящиеся к паттернам проектирования, чрезвычайно обширны, и все же большинство обсуждений вращается вокруг паттернов, представленных в книге. Нисколько не умаляя значения остальных паттернов, можно сказать, что 32 паттерна, описанные в книге, заслуживают звания «классики».

Впрочем, пристального внимания в книге заслуживают не только описания паттернов, но и вводная глава, в которой обосновываются многие идеи, а также проводится единая связующая нить. В этой главе описаны принципы объектно-ориентированного проектирования, рассчитанного на повторное использование. Второй принцип гласит, что композиция объектов (отношение «содержит») предпочтительнее наследования классов (отношение «является частным случаем»). Напомню, первый принцип рекомендует «программировать интерфейс, а не реализацию» – впрочем, это понятно любому, кто прочитал хоть что-то об инкапсуляции за последние 40 лет.

Программистам, не следившим за тем, как объектно-ориентированное программирование занимало центральное место в 1980-е и 1990-е годы, это правило может показаться не слишком важным. Но следует вспомнить, что в то время одной из определяющих концепций объектно-ориентированного программирования было наследование. Возьмем хотя бы описание C++ в книге Бьярна Страуструпа «The C++ Programming Language» (1985). Из него мы узнаем, что:

> C++ – язык программирования общего назначения, хорошо подходящий для системного программирования. Этот язык:
>
> - Лучше C
> - Поддерживает абстракции данных
> - Поддерживает объектно-ориентированное программирование
> - Поддерживает параметризацию типов

Если мы теперь посмотрим, что именно означает «поддерживает объектно-ориентированное программирование», то узнаем следующее:

Парадигма программирования определяется следующим образом:

- Решите, какие классы вам нужны
- Предоставьте полный набор операций для каждого класса
- Явно выразите общность посредством наследования

Для контраста возьмем другой, тоже классический труд Джошуа Блоха (Joshua Bloch) «Effective Java» (2008). В нем встречается не менее трех рекомендаций против наследования:

- Отдавайте предпочтение композиции перед наследованием.
- Проектируйте и документируйте наследование – или не используйте его вовсе.
- Отдавайте предпочтение интерфейсам перед абстрактными классами.

Означает ли это, что наследования следует избегать? Вопрос далеко не праздный. Программирование для Microsoft Windows может быть весьма неприятным делом (даже если осваивать его по очень приятной книге Чарльза Петцольда (Charles Petzold) «Programming Windows» [1999]). Когда появились первые инфраструктуры для Windows-программирования (разработки Borland и Microsoft), они казались глотком свежего воздуха. До этого даже создание простого окна было весьма непростой задачей. Чтобы программировать для Microsoft Windows, программисту приходилось работать с так называемым классом окна, который не имел ничего общего с классами C++. С новыми инфраструктурами было достаточно создать подкласс на основе класса, предоставляемого инфраструктурой... и все. Мы радовались внезапному освобождению от всей (или почти всей) рутины, а также внезапно найденному удачному применению объектно-ориентированных принципов.

Программирование для Microsoft Windows – всего лишь пример; энтузиазм по поводу объектно-ориентированного программирования и наследования был всеобщим. Странно видеть, что мы тогда могли ошибаться – хотя, вероятно, все же не ошибались. Наследование не является плохим по своей природе. Как и любая технология, оно может применяться как во благо, так и во вред. Плохие применения наследования были документированы во многих местах (хорошей отправной точкой служит та же книга «Банды Четырех»). В этой главе приводится пример красивой программной системы, построенной на основе наследования. Такой системой является Smalltalk.

Smalltalk принадлежит к числу чистых объектно-ориентированных языков. Хотя Smalltalk так и не пошел «в массы», он во многих отношениях повлиял на эволюцию современных языков. Возможно, по степени влияния на последующие языки программирования он сопо-

ставим с Algol-60 – еще один пример языка, который больше оказывал влияние, чем реально использовался.

Эта глава не является описанием языка программирования Smalltalk и его окружения (на самом деле они тесно связаны друг с другом). Скорее, в ней представлены основные архитектурные идеи, а также показано, куда они приводят нас в повседневной работе. Мы обсудим базовые принципы архитектуры и те подсознательные стимулы (affordances), которые они предоставляют программисту. Дональд Норман в своей книге «The Psychology of Everyday Things» (1988) доступно (и занимательно) объясняет концепцию подсознательных стимулов. Проще говоря, объект своим внешним видом разрешает (и даже приглашает нас) выполнить определенные действия. Висящая веревка приглашает нас дотянуться и дернуть за нее; горизонтальная рукоятка – нажать; дверная ручка – повернуть, и так далее. Аналогичным образом субъективное представление программиста о языке приглашает его выполнить определенные действия и предостерегает против выполнения других действий. Красиво построенный язык имеет красивую архитектуру, которая непременно отразится в написанных на нем программах.

Сильным выражением этого принципа является гипотеза Сапира–Ворфа (SWH), утверждающая, что язык определяет особенности мышления. Эта гипотеза в течение многих лет вызывала интерес у лингвистов и разработчиков языков. Предисловие к первому изданию «The C++ Programming Language» начинается с SWH, а в 1980 году К. Э. Иверсон (создатель языка APL) посвятил свою лекцию по случаю присуждения Премии Тьюринга важности обозначений для выражения наших мыслей. Истинность гипотезы Сапира–Ворфа небесспорна (в конце концов, каждому доводилось попадать в ситуацию, когда он не мог подобрать слова для выражения своих мыслей, так что мы думаем о большем, чем говорим), но в компьютерном коде связь между языками и программами очевидна. Мы знаем, что компьютерные языки обладают полнотой по Тьюрингу, но мы также знаем, что для некоторых областей одни языки подходят лучше других.

Однако архитектура языка интересна сама по себе независимо от ее влияния на архитектуру программ. Мы познакомимся с собственной архитектурой Smalltalk – с решениями реализации, концепциями проектирования и паттернами. Многие из них встречаются сегодня в более современных языках; а другие дают нам повод остановиться и поразмыслить о том, почему они не прижились.

От читателя не потребуется знание Smalltalk, хотя к концу главы будет рассмотрена значительная часть возможностей языка. Мы выделим основные архитектурные принципы и проиллюстрируем их небольши-

ми фрагментами кода. Одно из преимуществ сильной архитектуры заключается в том, что вам достаточно усвоить минимальное количество принципов, после чего вся логика инфраструктуры становится на свои места. Система Smalltalk, которая будет использоваться в описании, называется Squeak (*http://www.squeak.org*) – реализация с открытым кодом. Некоторые примеры трудно понять с первого прочтения, поскольку концепции излагаются относительно неформально, но они поясняются в следующих примерах. Постарайтесь и прочитайте до конца, а потом вернитесь к тому, что осталось непонятным.

В описании Smalltalk встретятся некоторые возможности, которые не поддерживаются в вашем основном языке программирования. Проблем с этим быть не должно. Один из проверенных временем принципов разработки гласит: чтобы использовать некоторую возможность при программировании, не обязательно, чтобы эта возможность была интегрирована на уровне языка. Приложив некоторые усилия, можно найти элегантное альтернативное решение на выбранном вами языке. В книге Стива Макконнелка (Steve McConnell) «Code Complete» (2004)[1] это называется *программированием в языке* (с. 69):

> Очень важно понимать различия между программированием на языке и программированием в языке... Многие важные принципы программирования зависят не от конкретных языков, а от способов их использования. Если в вашем языке отсутствуют конструкции, которые вам хотелось бы использовать или возникают другие проблемы, постарайтесь компенсировать недостатки. Изобретайте собственные конвенции, стандарты, библиотеки классов и другие усовершенствования.

Помнится, в 1990-е годы, когда объектно-ориентированный подход стал всеобщей нормой, в магазине технической литературы нам попалась книга по объектно-ориентированному ассемблеру – свидетельство творческого подхода (или упрямства) программистов, напрямую противоречившее SWH. Позднее Рэндалл Хайд (Randall Hide) объединил ассемблер с классами, наследованием и другими «украшениями» в языке HLA (High Level Assembler).

Мы подойдем к языку программирования так же, как к классической книге. Прежде чем серьезно браться за Smalltalk, начнем с некоторых определений из эссе Итало Кальвино (Italo Calvino) «Why Read the Classics» (1986):

> К классике относятся те книги, о которых люди обычно говорят «Я сейчас перечитываю...», и почти никогда «Я сейчас читаю»...

---

[1] Стив Макконнелл «Совершенный код», Питер, 2009.

> Словом «классика» мы называем книги, высоко ценимые теми, кто прочитал и полюбил их; но эти книги столь же высоко ценятся и теми, кому выпало счастье прочитать их впервые в условиях, обеспечивающих получение максимального удовольствия.
>
> Классические книги имеют странную силу – и когда они отказываются стираться из нашего разума, и когда они прячутся в складках памяти, маскируясь под коллективное или индивидуальное бессознательное.
>
> Перечитывая классику, вы каждый раз испытываете ту же радость открытия, как и при первом прочтении.
>
> Классическая книга никогда не перестает говорить то, что она должна говорить.
>
> Классические книги приходят к нам со следами предыдущих прочтений; когда они пробуждаются, мы видим следы, оставленные ими в культуре (или культурах), через которые они прошли (или, проще говоря, в языках и обычаях).
>
> Классика не всегда учит нас тому, чего мы не знали прежде. В классике мы иногда обнаруживаем то, что знали всегда (или думали, что знали) – но не подозревали о том, что именно этот автор впервые высказал эту мысль, или, по крайней мере, имеет к ней особое отношение. И это тоже становится для нас сюрпризом, от которого мы получаем много удовольствия, как всегда бывает при обнаружении неожиданного происхождения, логической связи или сродства.
>
> К классике относятся книги, которые дают нам еще больший заряд бодрости, открывают еще больше неожиданностей и чудес, чем мы могли предположить, когда слышали о них впервые.
>
> Классической является книга, написанная раньше других классических книг; но каждый, кто сначала прочитал остальные книги, а потом – эту, немедленно узнает ее место в семейном дереве.

Конечно, между книгами и языками программирования много различий, и все же эти определения могут применяться и к нашей задаче.

Объекты и только объекты

Популярные объектно-ориентированные языки наших дней (C++, Java и C#) не являются полностью объектно-ориентированными. Не все данные в них представляют собой объекты – некоторые относятся к *примитивным* типам. Например, вы не сможете объявить подкласс, производный от целого числа. Математические вычисления выполняются с обычными числами, без вызова методов объектов. Такой подход улучшает быстродействие, а также упрощает переход на объектно-ориентированный язык для людей с опытом работы на процедурных языках.

Но если оставить в системе только объектные данные, исключив примитивные типы, ситуация резко меняется. В Smalltalk целые числа длиной до 31 бита представляют собой экземпляры класса `SmallInteger` (более того, существует абстрактный класс `Integer` с подклассами `SmallInteger`, `LargePositiveInteger` и `LargeNegativeInteger`, а система автоматически выполняет преобразования по мере необходимости). С ними можно выполнять стандартные математические операции, но этим дело не ограничивается. Класс `SmallInteger` содержит более 670 методов (или *селекторов* в терминологии Smalltalk), что легко доказывает следующий фрагмент:

```
SmallInteger allSelectors size
```

Поучительно разобраться в том, как работает этот код. `allSelectors` – селектор класса, возвращающий все селекторы класса в виде множества `Set` (на самом деле `IdentitySet`, но сейчас это различие непринципиально). Множество `Set` само по себе является полноценным объектом со своими собственными селекторами; один из них, `size`, возвращает количество элементов, содержащихся в множестве.

Как и следовало ожидать, к числу селекторов `SmallInteger` принадлежат и математические операторы. Также в него входят тригонометрические и логарифмические функции, функции вычисления факториалов, наибольшего общего делителя и наименьшего кратного, функции для выполнения поразрядных операций и множество других операций.

Вместо целочисленных примитивов других языков в Smalltalk используются экземпляры `SmallInteger`. Это объясняет, почему работает конструкция:

```
2 raisedTo: 5
```

А еще интереснее – почему следующая конструкция тоже работает:

```
(7 + 3) timesRepeat: [ Transcript show: 'Hello, World'; cr ]
```

Если селектор вызывается с аргументами, то перед каждым аргументом ставится знак `:` (двоеточие). Исключение из этого правила составляют селекторы математических и логических операторов, как `+` в приведенном фрагменте. Класс `Transcript` представляет некий аналог системной консоли. Код `cr` обозначает возврат курсора (Carriage Return), а точка с запятой (`;`) разделяет сообщения, так что код `cr` передается объекту `Transcript`. Вы можете выполнить этот код в окне интерпретатора (в Smalltalk оно обычно называется *рабочей областью*) и посмотреть, что получится.

Конечно, не все 670 методов `SmallInteger` определяются в `SmallInteger`. Класс `SmallInteger` является частью иерархии наследования, показан-

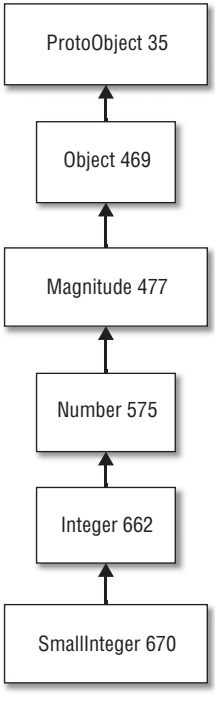


Рис. 14.1. *Иерархия SmallInteger*

ной на рис. 14.1; как видно из рисунка, многие селекторы определяются в предках `SmallInteger`. Большинство селекторов наследуется от `Object`, поэтому хорошее понимание функциональности `Object` многое объясняет в архитектуре Smalltalk (в Squeak фактическим корнем иерархии является класс `ProtoObject`, но это несущественно).

Экземпляры `Object` поддерживают селекторы сравнения – как равенства (`=`), так и тождественности (`==`); селекторы копирования – как глубокого (`deepCopy`), так и поверхностного (`shallowCopy`); селекторы вывода в потоки, обработки ошибок, отладки, обработки сообщений и т. д. Лишь немногие из сотен методов `Object` используются в повседневном программировании. Методы Smalltalk объединяются в группы, называемые *протоколами*; проверка описаний протоколов упрощает поиск методов.

Сами методы в Smalltalk также представляют собой полноценные объекты. Чтобы понять, что это означает в контексте общей архитектуры, возьмем следующий фрагмент кода:

```
aRectangle intersects: anotherRectangle
```

где aRectangle и anotherRectangle – экземпляры класса Rectangle. Когда aRectangle – *получатель* (так в Smalltalk называются объекты, получающие сообщения) – получает сообщение intersects:, интерпретатор Smalltalk выполняет следующие действия (Conroy and Pelegri-Llopart 1983):

1. Определение класса получателя.
2. Поиск селектора сообщения в классе и предках класса.
3. Выборка метода, связанного с селектором сообщения в том классе, в котором он был обнаружен.

В Smalltalk объектами являются не только числа, но и сами классы. Таким образом, SmallInteger, Object, Rectange – все это объекты. Когда интерпретатор ищет селектор сообщения в классе (шаг 2 в списке), он ищет его в содержимом объекта соответствующего класса. Точнее говоря, поиск производится в *словаре методов*. Словарь (экземпляр класса Dictionary) связывает ключи со значениями; словарь методов связывает каждый селектор с соответствующим экземпляром CompiledMethod.

Кстати говоря, intersects: элегантно реализуется на Smalltalk следующим образом:

```
(origin max: aRectangle origin) < (corner min: aRectangle corner)
```

Чтобы понять, почему эта конструкция работает, необходимо знать, что селектор origin возвращает левую верхнюю точку прямоугольника (экземпляр класса Point), селектор corner возвращает правую нижнюю точку прямоугольника, max: возвращает правый нижний угол прямоугольника, однозначно определяемого получателем и аргументом, а min: возвращает левый верхний угол прямоугольника, однозначно определяемого получателем и аргументом. Несмотря на свою отточенность, это выражение не оптимально; в Squeak используется альтернативное решение:

```
intersects: aRectangle
    "Answer whether aRectangle intersects the receiver anywhere."
    "Optimized; old code answered:
          (origin max: aRectangle origin) < (corner min: aRectangle corner)"

    | rOrigin rCorner |
    rOrigin := aRectangle origin.
    rCorner := aRectangle corner.
    rCorner x <= origin x ifTrue: [^ false].
    rCorner y <= origin y ifTrue: [^ false].
    rOrigin x >= corner x ifTrue: [^ false].
    rOrigin y >= corner y ifTrue: [^ false].
    ^ true
```

Работает быстрее, пусть и не так красиво. Этот фрагмент дает нам возможность представить читателю некоторые элементы синтаксиса Smalltalk. Переменные, локальные для метода, объявляются в блоке `| |`. Оператор присваивания имеет вид `:=`, знак `^` является эквивалентом `return` в C++ и Java, а точка (`.`) используется для разделения команд. Код в квадратных скобках (`[ ]`) называется *блоком*; это одна из ключевых концепций архитектуры Smalltalk. Блок является *замыканием*, то есть фрагментом кода, который может обращаться к переменным, определяемым в окружающей области видимости. Блоки в Smalltalk представлены классом `BlockContext`. Содержимое блока выполняется при получении объектом блока сообщения `value`, причем во многих случаях (как здесь) сообщение отправляется неявно. Комментарии в Smalltalk заключаются в кавычки; апострофы используются для определения строк.

Класс `BlockContext` наследует получателя, аргументы, временные переменные и отправителя создавшего его контекста. Существует аналогичный класс `MethodContext`, который представляет все динамическое состояние, связанное с выполнением метода (который, как мы видели, представлен массивом байт-кодов `CompiledMethod`). Как и следовало ожидать от объектно-ориентированного языка, `BlockContext` и `MethodContext` являются подклассами класса `ContextPart`. Класс `ContextPart` добавляет семантику исполнения в свой суперкласс `InstructionStream`. В свою очередь, экземпляры класса `InstructionStream` могут интерпретировать код Smalltalk. Суперклассом `InstructionStream` является класс `Object`, на котором цепочка наследования кончается.

Кроме селектора `value`, блоки также содержат селектор `fork`, реализующий параллелизм в языке. Так как в Smalltalk все является объектом, процессы являются экземплярами класса `Process`. Класс `Delay` позволяет приостановить выполнение процесса на заданный период времени; объект `Delay` приостанавливает текущий выполняемый процесс при получении сообщения `wait`. Объединив все сказанное, мы можем построить простейшую реализацию часов следующим образом (Goldberg and Robson 1989, p. 266):

```
[[true] whileTrue:
    [Time now printString displayAt: 100@100.
    (Delay forSeconds: 1) wait]] fork
```

Селектор `whileTrue:` выполняет свой аргумент-блок, пока его собственный блок-получатель остается истинным. Символ `@` – селектор класса `Number`, конструирующий экземпляры класса `Point`.

Блоки также предоставляют основную функциональность обработки ошибок. Суть в том, что вы указываете блоки, которые должны выпол-

няться при возникновении проблем. Например, в объекте Collection (контейнер для объектов) метод remove: пытается удалить из коллекции заданный элемент. Метод remove:ifAbsent: пытается удалить заданный элемент из коллекции, а если такой элемент не существует – выполняет блок, переданный в аргументе ifAbsent:. Ниже приведен пример удачной минимизации кода, где первая часть реализуется в контексте второй:

```
remove: oldObject
    "Remove oldObject from the receiver's elements. Answer oldObject
    unless no element is equal to oldObject, in which case, raise
    an error. ArrayedCollections cannot respond to this message."

    ^ self remove: oldObject ifAbsent: [self errorNotFound: oldObject]

remove: oldObject ifAbsent: anExceptionBlock
    "Remove oldObject from the receiver's elements. If several of the
    elements are equal to oldObject, only one is removed. If no element
    is equal to oldObject, answer the result of evaluating
    anExceptionBlock. Otherwise, answer the argument, oldObject.
    ArrayedCollections cannot respond to this message."

    self subclassResponsibility
```

Ключевое слово self представляет ссылку на текущий объект (аналог this в C++ и Java); это зарезервированное имя, *псевдопеременная* с фиксированной семантикой. Существуют и другие псевдопеременные: super – ссылка на суперкласс (аналог super в языке Java); nil, true и false пояснений не требуют; также существует псевдопеременная thisContext, которую мы видим в действии при определении метода subclassResponsibility. Этот селектор определяется в Object, и всего лишь указывает на необходимость переопределения в подклассе:

```
subclassResponsibility
    "This message sets up a framework for the behavior
    of the class's subclasses. Announce that the subclass
    should have implemented this message."

    self error: 'My subclass should have overridden ', thisContext
 sender selector printString
```

Псевдопеременная thisContext ссылается на текущий контекст выполнения (выполняемый метод или блок), то есть на экземпляр MethodContext и BlockContext. Селектор sender возвращает контекст, отправивший сообщение; selector возвращает селектор метода. *Все они тоже являются объектами.*

Концепция работы с кодом как с объектами не нова; сила Lisp проявляется в аналогичной работе с кодом и данными. В частности, она по-

зволяет нам программировать с использованием рефлексии, т. е. применять метапрограммирование.

Важность идеи метапрограммирования растет со временем. В компилируемых языках со статической типизацией, вроде C или C++, поддержка метапрограммирования незначительна. Попытка организовать поддержку метапрограммирования в C++ (Forman and Danforth, 1999), похоже, больше повлияла на Python, чем на сам C++ (см. PEP 253, «Subtyping Built-in Types», по адресу *http://www.python.org/dev/peps/pep-0253/*). Шаблонное метапрограммирование в C++ работает по другому принципу: мы используем тот факт, что компилятор C++ генерирует шаблонный код во время компиляции для его немедленной обработки (Abrahams and Gurtovoy, 2005). Эта методология открывает интересные возможности, которые, однако, требуют весьма серьезных навыков программирования. В Java метапрограммирование на базе рефлексии является неотъемлемой частью языка, хотя код Java, использующий рефлексию, обычно получается довольно неуклюжим.

Взаимосвязанная проблема проявляется при конструировании меню во время выполнения. Меню связывает визуальные элементы с обработчиками, которые должны активизироваться при выборе пользователем соответствующей метки. Если бы мы могли ссылаться на обработчики по имени, то меню можно было бы сконструировать динамически, используя конструкции следующего вида:

```
CustomMenu new addList: #(
    #('red' #redHandler)
    #('green' #greenHandler)
    #('blue' #blueHandler)); startUpWithCaption:'Colors'.
```

В Smalltalk ограничители `#( )` используются для определения массивов. Мы создаем новое меню с элементами и обработчиками по списку, содержащему пары «метка-обработчик». Обработчики представлены селекторами (в реальном коде нам пришлось бы определить для них реализацию). Префикс `#` в Smalltalk обозначает символические имена. Считайте, что это разновидность обычных строк. Символические имена обычно используются в именах методов и классов.

Рефлексия позволяет лаконично и выразительно реализовать паттерн «абстрактная фабрика» (Alpert et al. 1998, с. 43–44). Если мы хотим создать класс-фабрику, который создает объекты классов, задаваемых пользователем во время выполнения, это можно сделать так:

```
makeCar: manufacturersName
    "manufacturersName is a Symbol, such as #Ford, #Toyota,
    or #Porsche."
    | carClass |
```

```
    carClass := Smalltalk
                    at: (manufacturersName, #Car) asSymbol
                    ifAbsent: [^ nil].
    ^ carClass new
```

После того как пользователь укажет фирму-производителя, мы создаем имя класса, присоединяя к названию фирмы слово `Car`. Для фирмы `#Ford` класс будет называться `#FordCar`, для `#Toyota` — `#ToyotaCar`, и т. д. Конкатенация в Smalltalk обозначается запятой (,). Чтобы полученная в результате конкатенации строка снова стала символическим именем, мы вызываем ее метод `asSymbol`. Все имена классов в Smalltalk хранятся в словаре `Smallktalk`, уникальном экземпляре класса `SystemDictionary`. Обнаружив требуемое имя класса, мы возвращаем экземпляр этого класса.

Мы рассмотрели несколько примеров кода Smalltalk, но не определение класса. В Smalltalk классы конструируются тем же способом, как и все остальное: отправкой необходимого сообщения соответствующему получателю. Все начинается с заполнения следующего шаблона в среде Smalltalk:

```
NameOfSuperclass subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Unknown'
```

На место заполнителей `NameOfSuperclass` и `NameOfSubclass` подставляются настоящие имена. В секции `instanceVariableNames` перечисляются переменные экземпляров, в `classVariableNames` — переменные класса, а в секции `category` помечается категория, к которой относится наш класс (в Smalltalk классы группируются по категориям, по аналогии с пространствами имен или пакетами в других языках). В секции `poolDictionaries` перечисляются словари, используемые совместно с другими классами (так организуется совместное использование переменных в Smalltalk). После заполнения шаблон передается селектору `subclass` класса `Class`:

```
subclass: t instanceVariableNames: f classVariableNames: d
poolDictionaries: s category: cat
    "This is the standard initialization message for creating
    a new class as a subclass of an existing class (the receiver)."
    ^(ClassBuilder new)
          superclass: self
          subclass: t
          instanceVariableNames: f
          classVariableNames: d
```

```
        poolDictionaries: s
        category: cat
```

Селектор `subclass` создает экземпляр класса `ClassBuilder`, который создает новые или модифицирует существующие классы. Необходимая информация передается экземпляру `ClassBuilder`, чтобы новый класс был создан в соответствии с данными, введенными нами в шаблоне класса.

Выполнение всех действий с объектами посредством отправки сообщений сокращает количество концепций, которые необходимо усвоить при изучении Smalltalk. Кроме того, оно ограничивает количество синтаксических конструкций в языке. Стремление к минимализму в языках программирования имеет давнюю историю. В первой статье о Lisp (McCarthy, 1960) мы узнаем, что в Lisp существуют два класса выражений: s-выражения (или синтаксические выражения), строящиеся на основе списков, и m-выражения (метавыражения), использующие s-выражения в качестве данных. В конечном счете программисты выбрали одни лишь s-выражения, и Lisp стал таким, каким мы знаем его сегодня: языком, почти лишенным синтаксиса, в котором все (и программа, и данные) представляет собой списки. В зависимости от вашего отношения к Lisp это доказывает, что всего одной идеи достаточно для выражения даже самых сложных конструкций (или что человека можно заставить довольствоваться чем угодно).

Smalltalk не ограничивается одним синтаксическим элементом. В программах Smalltalk используются шесть разновидностей структурных элементов:

1. Ключевые слова, или псевдопеременные (их всего шесть: `self`, `super`, `nil`, `true`, `false` и `thisContext`).
2. Константы.
3. Объявления переменных.
4. Присваивание.
5. Блоки.
6. Сообщения.

Возможно, то, что отсутствует в этом списке, интереснее всего содержимого: мы не видим в нем никаких элементов для управления выполнением программы – ни условных конструкций, ни циклов. Эти конструкции не нужны, потому что они выражаются посредством сообщений, объектов и блоков (которые тоже являются объектами). Следующий метод реализует функцию вычисления факториала в классе `Integer`:

```
factorial
    "Answer the factorial of the receiver."
```

```
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

Операторы = и > представляют собой селекторы сообщений, возвращающие объекты абстрактного класса Boolean с двумя подклассами, True и False. Если получатель селектора ifTrue: является экземпляром True, то выполняется его аргумент – блок, заключенный в ограничители [ ]. Также существует симметричный селектор ifFalse: с противоположной семантикой. В общем случае рекурсию лучше заменять циклами; реализация функции с применением цикла выглядит так:

```
factorial
    "Implement factorial function using a loop"

    | returnVal |
    returnVal := 1.
    self >= 0
          ifTrue: [2
                    to: self
                    do: [:n | returnVal := returnVal * n]]
          ifFalse: [self error: 'Not valid for negative integers'].
    ^ returnVal
```

Основная часть работы выполняется в двух блоках. Первый блок выполняется для положительных значений, начиная с 2 и до значения получателя. Текущее значение каждой итерации передается внутреннему блоку, где вычисляется результат. Аргументы блоков снабжаются префиксом : (двоеточие) и отделяются от тела блока знаком | (вертикальная черта). Аргументов-блоков может быть несколько, как в следующем определении факториала (Black et al., 2007, с. 212):

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each |
                                                product * each ] ].
```

Селектор to: возвращает экземпляр класса Interval, который фактически перечисляет значения от 1 до 10. Для числа n блок factorial сначала присваивает аргументу product внутреннего блока значение 1. Далее он вызывает внутренний блок для каждого значения от 1 до n, умножая значение product на текущее число и сохраняя результат в product. Чтобы вычислить факториал 10, мы используем запись factorial value: 10. Герберт Саймон (Herbert Simon) в своей книге «The Sciences of the Artificial» (1996) цитирует голландского физика Саймона Стевина (Simon Stevin):

Wonder, en is gheen wonder (то есть: «Удивительно, но постижимо»).

Неявное определение типов

Хотя все в Smalltalk, даже классы, представляет собой объекты, классы не являются аналогами типов в понимании таких языков, как C++ или Java. Типы определяются неявно по тому, что они делают, и по их интерфейсам. В Smalltalk это называется *латентной типизацией*.

Латентная типизация является единственным механизмом типизации в Smalltalk (а также в ряде других языков с динамической типизацией), но это не означает, что она не играет важной роли в языках с сильной типизацией. Например, в C++ латентная типизация является основой параметризованного программирования с использованием шаблонов. Логично начать знакомство с ней с этого языка. Взгляните на следующий вводный пример использования шаблонов C++ (Vandervoorde and Josuttis, 2002, 2.4):

```
// Максимум из двух значений типа int
inline int const& max (int const& a, int const& b)
{
    return  a < b ? b : a;
}

// Максимум из двух значений любого типа
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return  a < b ? b : a;
}

// Максимум из трех значений любого типа
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);     // шаблон для трех аргументов
    ::max(7.0, 42.0);     // max<double> (на основании анализа аргументов)
    ::max('a', 'b');      // max<char> (на основании анализа аргументов)
    ::max(7, 42);         // нешаблонная версия для двух int
    ::max<>(7, 42);       // max<int> (на основании анализа аргументов)
    ::max<double>(7, 42); // max<double> (без анализа аргументов)
    ::max('a', 42.7);     // нешаблонная версия для двух int
}
```

Как видно из `main()`, функция `::max` будет работать для любого типа, реализующего оператор сравнения. В C++ этот тип может быть как примитивным, так и определяемым пользователем. Не существует ограничений, требующих, чтобы он был производным от конкретного класса. Допустим любой тип при условии, что он удовлетворяет базовому требованию о реализации сравнения. Неявное определение типа выглядит так: любой тип, для которого имеет смысл оператор `<`.

В школе нас учили, что существуют два способа определения множеств. Первый способ – явное перечисление элементов (*экстенсиональное определение*). Примером может послужить известное нам множество натуральных чисел: {1, 2, 3, ...}. Второй способ основан на описании общих признаков элементов множества (*интенсиональное определение*). Например, интенсиональное определение множества натуральных чисел может выглядеть так: «целые положительные числа». При объявлении объекта в C++ используется экстенсиональное определение: мы говорим, что объект принадлежит к заданному типу. Однако при использовании шаблонов фактически происходит переход к интенсиональному определению: мы говорим, что набор объектов, к которым применяется некоторый код, суть набор объектов, обладающих заданными свойствами (то есть предоставляющих необходимые операции).

К сожалению, в Java ситуация усложняется. Java поддерживает параметризованные шаблоны, но с шаблонами C++ их связывает разве что поверхностное синтаксическое сходство. Брюс Экель (Bruce Eckel) выразительно объяснил суть проблемы (см. *http://www.mindview.net/WebLog/log-0050*). В языках, поддерживающих латентную типизацию (таких как Python), можно сделать следующее:

```
class Dog:
    def talk(self):  print "Arf!"
    def reproduce(self): pass

class Robot:
    def talk(self): print "Click!"
    def oilChange(self): pass

a = Dog()
b = Robot()
speak(a)
speak(b)
```

Два вызова `speak()` работают независимо от типа аргумента. То же самое можно сделать и на C++:

```
class Dog {
public:
    void talk() { }
```

```
    void reproduce() { }
};

class Robot {
public:
    void talk() { }
    void oilChange() { }
};

template<class T> void speak(T speaker) {
    speaker.talk();
}

int main() {
    Dog d;
    Robot r;
    speak(d);
    speak(r);
}
```

Однако повторить то же самое на Java не удастся, поскольку следующий код не компилируется:

```
public class Communicate  {
    public <T> void speak(T speaker) {
        speaker.talk();
    }
}
```

Ситуация усложняется тем, что следующий фрагмент компилируется, поскольку во внутренней реализации параметризованные типы в языке java преобразуются в экземпляры `Object` (это называется *стиранием* (erasure)):

```
public class Communicate  {
    public <T> void speak(T speaker) {
      speaker.toString(); // Методы Object работают!
    }
}
```

В подобных ситуациях приходится действовать примерно так:

```
interface Speaks { void speak(); }

public class Communicate  {
    public <T extends Speaks> void speak(T speaker) {
      speaker.speak();
    }
}
```

Но такое решение во многом теряет все преимущества параметризации, поскольку мы определяем тип через интерфейс `Speak`. Отсутствие универсальности проявляется еще и в том, что примитивные типы Java не могут использоваться с механизмом параметризации. В качестве обходного решения Java предлагает *обертки* – полноценные классы, представляющие примитивные типы. Раньше преобразования между примитивами и обертками становились рутинной работой для Java-программиста. В последних версиях языка благодаря механизму автоупаковки в некоторых обстоятельствах преобразование выполняется автоматически. Как бы то ни было, в программе можно использовать запись `List<Integer>`, но не `List<int>`.

Недавно латентная типизация стала набирать популярность из-за ее широкого применения в языке программирования Ruby. Ее также называют «утиной типизацией» – шуточная отсылка к индуктивному умозаключению, приписываемому Джеймсу Уиткомбу Райли (James Whitcomb Riley):

> Если что-то ходит как утка и крякает как утка, то я буду считать это уткой.

Чтобы понять важность латентной типизации, рассмотрим одну из важнейших функций объектно-ориентированного программирования – *полиморфизма*. Этим термином обозначается использование разных типов в едином контексте. Один из вариантов реализации полиморфизма основан на применении наследования. Подкласс может использоваться (а вернее, должен использоваться, потому что программисты порой бывают невнимательны) всюду, где может использоваться суперкласс. Латентная типизация открывает дополнительную возможность для реализации полиморфизма: тип может использоваться везде, где он предоставляет методы, соответствующие контексту. В приведенном ранее примере на Python и C++ классы `Dog` и `Robot` *не имеют* общего суперкласса.

Конечно, в программах можно обойтись и без латентной типизации, ограничиваясь полиморфизмом на базе наследования. Однако разнообразие инструментов для решения возникающих проблем только идет на пользу программисту. Если обилие инструментов не становится препятствием в его работе, программист может выбрать среди них тот, который наилучшим образом подходит для данной ситуации. Эта идея была очень элегантно выражена Бьерном Страуструпом в книге «The Design and Evolution of C++» (1994, с. 23):

> Мой интерес к компьютерам и языкам программирования в основном имеет сугубо прагматическую природу.

> С эмпириками я чувствую себя увереннее, чем с идеалистами... Иначе говоря, я предпочитаю Аристотеля Платону, Хьюма Декарту и печально качаю головой над Паскалем. Всеобъемлющие «системы» наподобие тех, что встречаются у Платона и Канта, очаровательны, но они кажутся мне неудовлетворительными из-за своей опасной оторванности от повседневного опыта и особенностей конкретных личностей.
>
> Почти фанатичное внимание Киркегора к переживаниям отдельной личности и его проницательные психологические наблюдения кажутся мне гораздо более привлекательными, чем грандиозные планы и забота обо всем человечестве в абстракциях Гегеля или Маркса. Уважение интересов групп, которое не подразумевает уважения интересов отдельных личностей в этих группах, ничего не стоит. Многие архитектурные решения C++ уходят своими корнями в мое нежелание заставлять людей делать что-то одним определенным способом. В истории самые страшные катастрофы происходили из-за идеалистов, которые пытались заставить людей «делать то, что им же пойдет во благо». Такой идеализм не только приводит к страданиям невинных жертв, но и создает ложные представления и развращает самих идеалистов, применяющих силу. Я также нахожу, что идеалисты склонны игнорировать неудобный практический опыт, который противоречит их догмам или теориям. Там, где возникают конфликты идей – а иногда даже там, где ведущие специалисты вроде бы приходят к единому мнению – я предпочитаю действовать так, чтобы предоставить программисту выбор.

Возвращаясь к Smalltalk, рассмотрим задачу перебора коллекции объектов, применения функции к каждому объекту и сбора результатов. Реализация выглядит следующим образом:

```
collect: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Collect the resulting values into a collection like the receiver.
    Answer the new collection."

    | newCollection |
    newCollection := self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ^ newCollection
```

Чтобы понять этот метод, необходимо знать, что метод `species` возвращает либо класс получателя, либо класс, сходный с ним, – различия слишком тонкие, чтобы обращать на них внимание. Нас интересует то, что для конструирования новой коллекции нам необходим только объект с селектором `value`, который мы вызываем. Блоки имеют селектор с именем `value`, поэтому для конструирования может использоваться любой блок. Впрочем, это могут быть не только блоки; подойдет все, что реализует `value`.

Концепция «чего угодно, возвращающего значение» оказалась настолько важной, что ей было присвоено собственное имя: она называется *объектом-функцией* и является одной из фундаментальных составляющих алгоритмов C++ STL. Функция, которая перебирает элементы списка, применяет преобразование и возвращает список результатов, традиционно применялась в функциональном программировании (где обычно используется термин *функция-отображение*). Разумеется, данная возможность существует в Lisp. Она также предусмотрена в Python, что позволяет использовать конструкции следующего вида:

```
def negate(x): return -x
map(negate, range(1, 10))
```

тогда как в Perl мы бы написали:

```
map { -$_ } (1..10)
```

Эквивалентную операцию можно выполнить и в C++ STL (Josuttis 1999, 9.6.2):

```
vector<int> coll1;
list<int> coll2;

// инициализация coll1

// инвертирование всех элементов coll1
transform(coll1.begin(), coll1.end(),  //исходный диапазон
          back_inserter(coll2),        //приемный диапазон
          negate<int>());              //операция
```

Как ни трагично, многие программисты C++ записывают этот код с использованием циклического перебора элементов массива.

У латентной типизации есть не только преимущества, но и недостатки. В языках со статической типизацией (таких как C++) компилятор проверяет, что объект, используемый в выражении с латентным типом, действительно обладает необходимым интерфейсом. В языках с динамической типизацией, в том числе и в Smalltalk, несоответствия обнаруживаются во время выполнения, при выдаче ошибки.

Просто проигнорировать эти проблемы нельзя. Языки с сильной типизацией предотвращают просчеты программистов; они особенно полезны в крупных проектах, в которых хорошая структура упрощает сопровождение. Самым важным изменением при переходе от традиционного UNIX C к ANSI C в 1980-е годы стало усиление системы типов: в C появились прототипы функций, а аргументы функций стали проверяться во время компиляции. Теперь мы неодобрительно относимся к лихим преобразованиям типов. Короче говоря, мы отказались от

части свободы ради дисциплины, а вернее, поменяли хаос на некоторый порядок.

В Smalltalk обычно предполагается, что хаоса быть не должно, так как мы пишем небольшие фрагменты кода одновременно с их тестированием. В Smalltalk тестирование выполняется просто. Четкие различия между циклами компиляции и построения отсутствуют, поэтому мы можем писать маленькие фрагменты кода в рабочей области и немедленно видеть, как себя ведет наш код. Упрощается и модульное тестирование; мы можем писать модульные тесты до кода, не беспокоясь о сообщениях компилятора о необъявленных типах или методах. В сообщество, которое дало нам JUnit, входит немало участников из сообщества Smalltalk; возможно, это неслучайно. Большая часть этих возможностей может быть реализована на Java с применением сценарных средств, например BeanShell или Groovy. В сущности, языки с сильной типизацией порождают в нас ложное чувство безопасности:

> Если программа, написанная на языке с сильной, статической типизацией, успешно компилируется, это означает лишь то, что она прошла некоторые проверки. Это означает, что программа заведомо не содержит синтаксических ошибок... Но тот факт, что ваш код успешно прошел компиляцию, еще не гарантирует его правильности. Если ваш код работает, это тоже не гарантирует его правильности.
>
> Единственной гарантией правильности независимо от того, имеет ли язык сильную или слабую типизацию, является прохождение им всех тестов, определяющих правильность программы. (*http://www.mindview.net/WebLog/log-0025*)

Впрочем, иногда осторожность приносит свои плоды. Допустим, мы добавляем объекты в коллекцию и ожидаем, что эти объекты обладают определенным интерфейсом. Было бы неприятно обнаружить во время выполнения программы, что некоторые объекты этому условию не удовлетворяют. Для защиты от подобных неприятностей можно воспользоваться средствами метапрограммирования Smalltalk.

Метапрограммирование дополняет латентную типизацию; оно позволяет нам убедиться в том, что указанный интерфейс действительно поддерживается объектом, и предпринять необходимые действия в противном случае – например, передавая нереализованные вызовы делегатам, которые их заведомо реализуют, или просто корректно обрабатывая ошибку с предотвращением фатального сбоя. В «книге с киркой» (Thomas et al. 2005, с. 370–371) приводится пример для языка Ruby, в котором в строку добавляется информация о песне:

```
def append_song(result, song)
  # Проверка правильности переданных параметров
```

```
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end
  result << song.title << " (" << song.artist << ")"
  end
```

Так мы бы действовали в стиле программирования Java или C#. В стиле Ruby с латентной типизацией будет достаточно следующего фрагмента:

```
def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end
```

Код работает с любым объектом, поддерживающим присоединение с использованием <<; для остальных объектов выдается исключение. Чтобы действительно застраховаться от возможных неприятностей, проверять следует возможности объекта, а не его тип:

```
def append_song(result, song)
  # Проверка правильности переданных параметров
  unless result.respond_to?(:<<)
    fail TypeError.new("`result' needs `<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("`song' needs 'artist' and 'title'")
  end
  result << song.title << " (" << song.artist << ")"
end
```

В Smalltalk имеется метод `respondsTo:`, определяемый в классе `Object`, при помощи которого можно во время выполнения проверить наличие конкретного селектора у конкретного получателя:

```
respondsTo: aSymbol
    "Answer whether the method dictionary of the receiver's class
    contains aSymbol as a message selector."

    ^self class canUnderstand: aSymbol
```

Реализация тривиальна: проверка делегируется селектору `canUnderstand:`, определяемому в `Behavior`:

```
canUnderstand: selector
    "Answer whether the receiver can respond to the message whose
    selector is the argument. The selector can be in the method
    dictionary of the receiver's class or any of its superclasses."
```

```
    (self includesSelector: selector) ifTrue: [^true].
    superclass == nil ifTrue: [^false].
    ^superclass canUnderstand: selector
```

Наконец, селектор `includesSelector:` также определяется в `Behavior`, где он ограничивается проверкой словаря методов класса:

```
includesSelector: aSymbol
    "Answer whether the message whose selector is the argument
    is in the method dictionary of the receiver's class."

    ^ self methodDict includesKey: aSymbol
```

Когда получатель получает сообщение, которое он не может понять, его стандартным ответом будет отправка системе сообщения `doesNotUnderstand:`. Если вы предпочитаете попытаться исправить ошибку самостоятельно, достаточно переопределить сообщение, что делается примерно так:

```
doesNotUnderstand: aMessage
    "Handles messages not being understood by attempting to
    proxy to a target"
    target perform: aMessage selector withArguments: aMessage arguments].
```

Предполагается, что `target` ссылается на объект-посредник, который, как мы надеемся, сможет обработать ошибочно направленное сообщение.

Проблемы

Открытое (public) наследование выражает отношения «является частным случаем». Построение иерархий классов, действительно соответствующих этому паттерну, требует тщательного планирования со стороны программиста. Если у вас имеется класс с методом и подкласс, в котором этот метод не имеет смысла, значит, отношения между классами не вписываются в схему открытого наследования, и его применение является признаком плохого проектирования. Впрочем, языки позволяют уладить эту проблему.

В C++ бессмысленный метод может возвращать код ошибки или инициировать исключение. Классический пример с птицами (Meyers, 2005, раздел 32):

```
class Bird {
public:
    virtual void fly();   // птицы умеют летать
    // ...
};
```

```
class Penguin: public Bird {    // пингвины - птицы
public:
    virtual void fly() { error("Attempt to make a penguin fly!"); }
    // ...
};
```

Программисты C++ также могут скрыть неподходящий метод:

```
class Base {
public:
    virtual void f()=0;
};

class Derived: public Base {
private:
    virtual void f() {
    }
};
```

Класс `Derived` уже не является абстрактным, но содержит совершенно бесполезную функцию `f()`. Подобных ухищрений следует избегать.

В Java также можно обойти проблему, возвращая код ошибки или инициируя исключение. Также можно объявить бессмысленный метод абстрактным в подклассе; в этом случае иерархия классов становится абстрактной от этой точки и до места определения конкретного метода. Такие ухищрения тоже нежелательны.

Обычным решением проблемы неподходящих или бессмысленных методов в иерархии классов является повторное проектирование этой иерархии. В иерархии классов, которая лучше отражает особенности мира пернатых, вводится подкласс `FlyingBird` для представления летающих птиц, а класс `Penguin` делается производным от `Bird`, а не от `FlyingBird`.

В Squeak всего 45 методов отправляют сообщение `shouldNotImplement`, используемое в тех случаях, когда метод, унаследованный от суперкласса, неприменим в текущем классе. Это весьма небольшая часть общего количества методов и объектов в Smalltalk, так что язык не отягощен плохо спроектированными иерархиями классов. Впрочем, даже сообщение `shouldNotImplement` фактически является реализацией. Это обстоятельство дает представление о более глубокой проблеме Smalltalk: в языке нет полноценных абстрактных классов или методов. Абстрактность методов условна – абстрактным считается метод, не имеющий никакой реализации.

Вместо того чтобы использовать сообщение `shouldNotImplement`, мы могли указать, что реализация метода является обязанностью подкласса;

как мы уже видели, именно для этой цели предназначено сообщение subclassResponsibility. В этом случае класс, из которого отправляется subclassResponsibility, по соглашению считается абстрактным. Например, класс Collection представляет обобщенный интерфейс добавления и удаления объектов, но не предоставляет реализации, поскольку реализация зависит от конкретного подкласса (словарь, массив, список...) Метод add: должен реализовываться в подклассах:

```
add: newObject
    "Include newObject as one of the receiver's elements. Answer newObject.
    ArrayedCollections cannot respond to this message."

    self subclassResponsibility
```

«Абстрактное» определение add: даже позволяет определять в Collection методы, использующие его (например, add:withOccurrences:):

```
add: newObject withOccurrences: anInteger
    "Add newObject anInteger times to the receiver. Answer newObject."

    anInteger timesRepeat: [self add: newObject].
    ^ newObject
```

При этом даже не нужно определять add:; add:withOccurrences: все равно определяется так, как показано, и Smalltalk не будет протестовать, если во время выполнения у объекта-получателя определен селектор add: (кстати говоря, add:WithOccurrences: — элегантная реализация паттерна «стратегия»). В то же время комментарий в add: указывает на то, что некоторые подклассы Collection, производные от подкласса ArrayedCollection, вообще не должны реализовывать сообщение. Это требование тоже устанавливается только во время выполнения при помощи shouldNotImplement:

```
add: newObject
    self shouldNotImplement
```

В использовании условностей при программировании нет ничего плохого; хорошее владение ими является одной из составляющих искусства программирования. Однако проблемы могут возникнуть из-за того, что достижение желаемого результата зависит исключительно от коллекций. Smalltalk не предупредит нас, если мы забудем о том, что add: в ArrayedCollection определять не нужно. Все кончится позорной ошибкой времени выполнения.

Ранее вы уже видели, как легко в Smalltalk реализуются классы-посредники. Но если вам нужен класс-посредник только для небольшого количества методов, ситуация усложняется. Причина кроется в отсутствии полноценных абстрактных классов. Класс-посредник может

быть подклассом того класса, который он должен представлять; в этом случае он наследует все методы такого класса, а не только те методы, которые он опосредует. Также можно применить латентную типизацию и определить в посреднике только опосредованные методы; но раз в системе нет ничего, кроме объектов, класс-посредник унаследует все методы класса `Object`. В идеале нам хотелось бы, чтобы класс, опосредующий два метода, содержал только эти два метода, но как это сделать – неясно. Класс будет содержать все методы, унаследованные им от предков. Можно пуститься на всевозможные хитрости для сокращения количества наследуемых методов; например, в некоторых диалектах Smalltalk можно подклассировать новый класс от `nil` вместо `Object`. В этом случае не наследуется ничего, но вам придется копировать и вставлять необходимые методы из `Object` (Alpert et al. 1998, с. 215). В Squeak классы также могут создаваться подклассированием `ProtoObject` вместо `Object`.

Вопрос о том, что откуда наследуется, весьма нетривиален. Так как в Smalltalk все, в том числе и классы, является объектами, возникает резонный вопрос – экземплярами чего являются классы? Оказывается, классы являются экземплярами *метаклассов*; имя метакласса образуется из имени класса и слова `class`. Например, метакласс класса `Object` называется `Object class`. Такая схема выглядит логично, но не отвечает на исходный вопрос – из нее неясно, экземплярами чего являются метаклассы. Оказывается, метаклассы являются экземплярами класса `Metaclass` (без суффикса `class`). А `Metaclass` является экземпляром другого класса с именем `Metaclass class` (с суффиксом `class`), который тоже должен быть экземпляром чего-то, поэтому он был сделан экземпляром `Metaclass` (если у читателя возникает впечатление, что он читает фрагмент сценария «12 обезьян» Терри Гиллиама, это вполне простительно). Ситуация еще сильнее усложняется, если принять во внимание отношения наследования (до настоящего момента речь шла только об экземплярах). Оказывается, `Object class` является подклассом `Class`, который является экземпляром `Class class`; `Class` является подклассом `ClassDescription`, производного от `Behavior`, производного от `Object`, а в Squeak на вершине иерархии находится класс `ProtoObject`. Ситуация графически представлена на рис. 14.2 (классы уровня ниже `Object` не показаны). Читатель может попытаться объединить рис. 14.1 с рис. 14.2 (и не забудьте добавить метаклассы, начиная со `SmallInteger class`). Как говорится, «слишком много хорошего тоже плохо»; хорошо, когда в системе нет ничего, кроме объектов, но выполнение всех последствий этой аксиомы не обязательно приводит к доступной структуре. К счастью, большинству программистов Smalltalk не нужно беспокоиться о подобных проблемах.

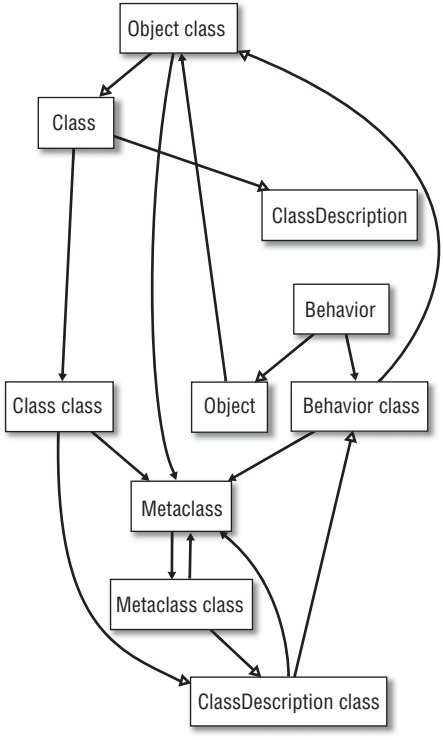


***Рис. 14.2.** Спагетти à la Smalltalk*

Однако у принципов «везде только объекты» и «все происходит посредством передачи сообщений» имеются и другие последствия, которые касаются рядовых программистов. Любой программист, даже с минимальным опытом, будет ожидать, что

```
3 + 5 * 7 == 38
```

Но в Smalltalk это не так:

```
3 + 5 * 7 == 56
```

Дело в том, что бинарные арифметические операторы по сути представляют собой обычные селекторы числовых классов. Соответственно они рассматриваются точно так же, как любые другие сообщения, и ничто не требует, чтобы одно бинарное сообщение обладало более высоким приоритетом по сравнению с другим. Правда, когда вы усво-

ите правила игры, это выглядит логично... и все-таки непривычно. Проверить результат математических вычислений в рабочей области среды Smalltalk несложно, но было бы лучше, если бы программисту не приходилось задумываться над подобными вещами.

Сама среда Smalltalk является одним из величайших новшеств Smalltalk. В 1980-е годы, когда графические дисплеи были редкостью, а для программирования в основном использовались монохромные текстовые терминалы, Smalltalk реализовал графический интерфейс для языка, построенного на базе виртуальной машины. Он определенно опередил свое время. Только спустя еще 10 лет виртуальные машины вошли в массовое программирование благодаря Java. Графические интерфейсы победили, но только после снижения цен на оборудование. В те времена среда с такими требованиями считалась непрактичной (да вероятно, таковой и была). Впрочем, и это еще не все.

Среда Smalltalk фактически представляет собой экосистему классов, с которыми вы не только можете работать, но и с которыми вам *приходится* работать. Существует только один разумный способ создания нового класса в Smalltalk: вы используете подходящее средство просмотра для поиска нужного класса, на основе которого будет создаваться подкласс. Вы привыкли к своему любимому редактору и отточили навыки сборки в командной строке на нескольких языках, но в Smalltalk дело обстоит иначе. Возможно, вам понравится библиотека классов, которую проектировщики Smalltalk предоставили вам, но если вы думаете иначе – ничего не поделаешь. Когда вы находитесь внутри среды Smalltalk, все хорошо, но чтобы что-то сделать, необходимо оказаться внутри. Не все программисты готовы к такой постановке вопроса. Кто-то из читателей после нескольких начальных страниц поторопится загрузить и установить Smalltalk – и в отчаянии вскинет руки, когда увидит, насколько чужим и незнакомым стало все вокруг.

Отчасти это объясняет, почему Smalltalk так и не завоевал широкой популярности. Smalltalk непреклонен; он не признает компромиссов. Он определил новую модель программирования, использовал концепции, которые позднее были приняты во многих других языках, во многих отношениях послужил образцом для подражания. В этом отношении он мало чем отличается от мира архитектуры.

Архитектура в камне

Вероятно, самым знаменитым домом Америки является «Дом над водопадом», спроектированный Фрэнком Ллойдом Райтом в 1935 году, –

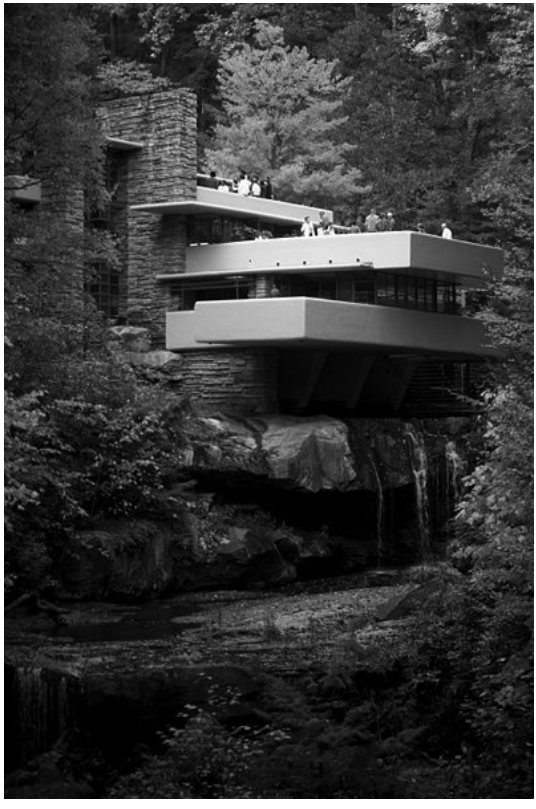


Рис. 14.3. *Дом над водопадом*

«вероятно, изображаемый на иллюстрациях чаще всех домов XX века» (Nuttgens, 1997, с. 264). Дом построен над водопадом в ущелье Беар-Ран для Эдгара Кауфмана старшего, миллионера из Питтсбурга. С 1937 по 1963 год он использовался как загородный дом Кауфманов, затем был пожертвован Природоохранному комитету Западной Пенсильвании и открылся для публики как музей в 1964 году.

Даже фотографии (рис. 14.3) передают впечатление безмятежности и красоты здания и его окружения. Райт стремился к слиянию природы с архитектурой, он хотел, чтобы искусство и природа отражали друг друга; виды природы украшают «дом над водопадом» изнутри, а сам дом становится ее частью. Этот шедевр современной архитектуры предлагает нам поразмыслить над тем значением, которое архитектор хотел вложить в свое произведение.

Однако дом – не только объект для восхищения, но и место, в котором живут. Мы не можем жить в «доме над водопадом» (в лучшем случае вам удастся только посетить его), но возможно, вы представляете себе, как будет замечательно там пожить. Весьма вероятно, что вы ошибаетесь! Из книги Стюарда Бренда (Steward Brand) «How Buildings Learn» (1995, с. 58) мы узнаем, что:

> Поздний шедевр Райта, «Дом над водопадом» в Пенсильвании, по результатам опроса AIA признанный «лучшим творением американской архитектуры всех времен», страдает от бесчисленных протечек, которые портят окна и каменные стены и разрушают несущие конструкции. Исходный владелец называл «дом над водопадом» «рассадником плесени», «семиведерным зданием». Дом действительно великолепный и впечатляющий, но жить в нем невозможно». (Цитата из Джудит Донахью (Judith Donahue), «Fixing Fallingwater's Flaws», Architecture, Nov. 1989, с. 100.)

Возможно, мнение излишне резкое. Эдгар Кауфман младший, живший в «доме над водопадом», описывал ситуацию немного иначе:

> При возведении «Дома над водопадом» было допущено немало ошибок, однако выдающаяся красота дома и эстетическое удовольствие, которое он приносил своим обитателям, формирует контекст для оценки сооружения. Жизнь в «Доме над водопадом» включала трудности и усилия по их преодолению (Kaufmann, 1986, с. 49).

К этому суждению тоже следует относиться с долей скепсиса. Эдгар Кауфман младший не был незаинтересованной стороной. В 1934 году он вступил в основанное Райтом «Сообщество Талиесина» (в 1932 году 23 ученика приехали жить и учиться в резиденции «Талиесин» в Спринг Грин, штат Висконсин; так была основана существующая до сих пор «Школа архитектуры Фрэнка Ллойда Райта»; сейчас школа имеет две площадки, «Талиесин» и «Западный Талиесин» в Аризоне). Он представил архитектора своим родителям и часто становился посредником между Райтом и своими родителями во время проектирования и строительства здания.

Другим архитектурным чудом – и, возможно, самым влиятельным домом XX века (влиятельным в том смысле, что его стиль определил морфологию современных городов по всему миру) – является Вилла Савой в Пуасси, пригороде Парижа, спроектированная швейцарским архитектором Шарлем-Эдуаром Жаннере-Гри, известным под псевдонимом Ле Корбюзье (рис. 14.4). Вилла Савой, как и «Дом над водопадом», проектировалась как загородный дом. Строительство велось с 1928 по 1931 год.

Рис. 14.4. *Вилла Савой*

У Райта были свои твердые представления о роли архитектуры; Ле Корбюзье тоже обладал твердыми убеждениями, но руководствовался несколько иными идеалами. Если Райт работал над отношениями искусства и природы, то:

> Ле Корбюзье изобрел систему пропорций «Модулор», объединявшую Золотое сечение, шестифутовую человеческую фигуру и гармонические пропорции в тщательно проработанной теоретической системе, которая, как предполагалось, будет сочетать механизацию с «естественным порядком» (Curtis, 1996, с. 412)

Вилла Савой вызывала священный трепет у архитекторов:

> Как и любая работа высокого порядка, Вилла Савой не поддается поверхностной классификации. Она проста и сложна, рассудительна и чувственна. Насыщенная идеями, она выражает их посредством форм, объемов и пространств, «находящихся в определенных отношениях». Являясь «классическим» произведением современной архитектуры, она также обладает сродством с архитектурой прошлого. Одним из краеугольных камней философии Ле Корбюзье было представление современной жизни в архитектурных формах непреходящей ценности, и в Вилле Савой мы легко узнаем отголоски старых классических тем: гармония, пропорции, ясность, простые антаблементы. (Curtis, 1996, с. 284)

Люди могут и не согласиться с такой оценкой; кому-то Вилла Савой может показаться белой коробкой, приготовленной к погрузке. В конце концов, *de gustibus et coloris non est diputandum*. Но о другом аспекте – практической ценности этого строения как дома – можно судить более определенно. Клиенты Ле Корбюзье смотрели на свой дом с иной точки зрения:

> Несмотря на начальные протесты семейства Савой, Ле Корбюзье настоял – предположительно на основании исключительно технических и экономических соображений, – что плоская крыша предпочтительнее скатной. Он заверил своих клиентов, что плоская крыша обойдется дешевле, будет более простой в эксплуатации и прохладной летом, а мадам Савой сможет заниматься на ней гимнастикой, не беспокоясь о влажности на уровне первого этажа. Но всего через неделю после переезда над спальней Роджера [сын четы Савой] произошла протечка. Воды было так много, что мальчик заболел бронхитом, который перешел в пневмонию, и в конечном итоге ему пришлось целый год восстанавливать здоровье в санатории в Шамони (De Botton, 2006, с. 65).

Выглядит как грубая шутка архитектора над клиентами, но, похоже, архитектор едва ушел от ответственности:

> В сентябре 1936 года, спустя шесть лет после официального завершения строительства, мадам Савой выразила свои чувства по поводу плоской крыши в письме (усеянном брызгами): «В холле льется вода, в переходе льется вода, а стена гаража полностью промокла. Более того, вода льется и в моей ванной; в плохую погоду там происходит потоп, так как вода проникает через люк в крыше». Ле Корбюзье обещал, что проблема будет немедленно устранена, и воспользовался случаем напомнить клиенту о том, с каким энтузиазмом плоская крыша была воспринята архитектурными критиками по всему миру: «Вам следовало бы положить на столик в нижнем холле книгу отзывов и попросить ваших посетителей оставлять в ней свои имена и адреса. Увидите, сколько редких автографов вы сможете собрать». Но идея коллекционирования автографов мало утешала семью Савой, страдающую от ревматизма. «После бесчисленных требований с моей стороны вы наконец-то согласились с тем, что дом, построенный в 1929 году, не пригоден для проживания», – укоряет мадам Савой осенью 1937 года. – «Речь идет о вашей репутации, и я не собираюсь оплачивать ремонт. Пожалуйста, немедленно сделайте мой дом годным для жилья. Я искренне надеюсь, что мне не придется прибегать к судебному иску». Только начало Второй мировой войны и бегство семьи Савой из Парижа избавило Ле Корбюзье от необходимости отвечать в суде за архитектуру своего непригодного для жизни, хотя и обладающего выдающейся красотой творения (De Botton, 2006, с. 65–66).

Другая культовая фигура современной архитектуры, Людвиг Мис ван дер Роэ, использовал минималистскую формальную систему, основан-

ную на двутавровых балках. Он «держал рядом со своим столом полноразмерные фрагменты двутавровых балок, чтобы точно представлять себе все пропорции. Он полагал, что двутавровая балка была современным эквивалентом дорической колонны» (Jencks, 2006, с. 13).

Мис ван дер Роэ больше известен благодаря своему принципу «меньше, значит больше», которым он хотел вернуть архитектуру к исходной сущности – лишить ее орнаментов, украшений или любых избыточных элементов, не имеющих функциональной цели. Двутавровые балки являются частью несущей конструкции здания.

Или так кажется. Одной из важнейших работ Миса ван дер Роэ стало здание Сигрэм, завершенное в 1958 году (рис. 14.5). Архитектор столкнулся с парадоксальной ситуацией: ему очень хотелось, чтобы двутавровые балки были видны в строении, однако это было невозможно по американским строительным нормам, которые требовали, чтобы ме-

Рис. 14.5. *Здание Сигрэм*

таллические несущие конструкции были покрыты огнеупорным материалом (например, цементом). Как ни удивительно, при внимательном взгляде на здание Сигрэм двутавровые балки все же видны. Это не настоящие несущие балки. Мис ван дер Роэ разместил на поверхности бутафорские балки, чтобы они «раскрывали» нижележащую структуру. Более того, чтобы зрительный образ здания не пострадал, жалюзи могли находиться только в трех положениях: открытом, закрытом и полузакрытом – пожалуй, не лучшая защита от солнечного света (Wolfe 1982, с. 75–76).

Ведущий архитектор современности Луис Салливан (среди прочего один из создателей концепции высотных зданий и учитель Райта) писал:

> Будь то орел, рассекающий крыльями воздух в полете, или распустившийся цветок яблони, или едва передвигающая ноги рабочая лошадь, или беспечный лебедь, или раскидистый дуб с петляющим у его подножия ручьем, или проплывающие мимо облака, или солнце, изливающее свет на все вокруг, *функция всегда определяет форму*, и это закон. Если функция остается неизменной, форма также не изменяется. Гранитные скалы, вечно задумчивые холмы не меняются веками; молния же рождается, обретает форму и гибнет в одно мгновение.
>
> Это всеобщий закон всех вещей – органических и неорганических, физических и метафизических, человеческих и сверхчеловеческих, всех истинных проявлений разума, сердца и души: жизнь проявляется в своих выражениях, а функция всегда определяет форму. Это закон. (Sullivan, 1896)

И после этого Мис ван дер Роэ превращает принцип «Функция определяет форму» в его полную противоположность. А может быть, такие афоризмы призваны стимулировать работу мысли, а не описывать то, что происходит на самом деле.

Пол Рэнд считается ведущим графическим дизайнером Америки. Ему принадлежат логотипы IBM, AB и исходный логотип UPS; он сотрудничал со Стивом Джобсом в NeXT Computer и написал влиятельные книги по теории дизайна. Он утверждает:

> Как было неоднократно показано, отделение формы от функции, концепции от исполнения редко приводит к созданию объектов, обладающих эстетической ценностью. Аналогичным образом любая система, рассматривающая эстетику как нечто несущественное, отделяющая художника от его творения, измельчающая работу личности или творческого коллектива, превращающая творческий процесс в ничто, – в долгосрочной перспективе обрекает на неудачу не только продукт, но и его создателя». (Rand, 1985, с. 3)

Современную архитектуру легко ругать. Работы Миса ван дер Роэ и Ле Корбюзье бездарно копировались по всему свету, и на них лежит ответственность за многие уродливые районы, криминогенные кварталы с дешевым жильем для рабочих, бездушные бизнес-центры. Интереснее присмотреться к критике самих создателей. Райт, Ле Корбюзье и Мис ван дер Роэ – все они подвергались критике за свою непреклонность; их критиковали за твердость, нежелание идти на компромисс. Благодаря их экстравагантным взглядам были созданы красивые здания – но не здания, которые обеспечивают нам материальный комфорт.

Непреклонность не всегда является пороком. В интервью для «Doctor Dobb's Journal» в апреле 1996 г. Дональда Кнута спросили, что он думает об Эдгаре Дейкстре. «Его величайшая сила, – ответил Кнут, – кроется в его бескомпромиссности. От одной мысли о программировании на C++ ему станет плохо». Дейкстра бескомпромиссен до такой степени, что он не притрагивался к компьютеру в течение нескольких лет, а потом написал свое замечательное эссе «Humble Programmer» (Смиренный программист) с обсуждением этой темы. Он был одним из самых влиятельных ученых в области компьютерных технологий; его труды все еще живы, они дают нам ценные советы, их полезно читать под натиском последней моды или очередной «серебряной пули» от программирования. Его непреклонная позиция только повысила ценность его работ для тех программистов, которые пишут программы для реальных компьютеров.

Возможно, это обстоятельство является ключом для понимания роли, которую Smalltalk (а перед ним Algol) играет в нашей профессиональной жизни. Некоторые архитекторы прокладывают новые пути и создают монументы для будущих поколений; по своей природе такие здания в большей степени скорее являются манифестами, чем домами или офисами. Бесспорно, «Дом над водопадом» производит впечатление на посетителей и вдохновляет молодых архитекторов – даже если мы считаем, что жилье из него вышло неважное... или после яростной защиты правила «функция определяет форму» незаметно украшаем строение. Нечто похожее происходит и в мире программирования: у некоторых систем лучше получалось влиять на пути развития программирования, чем использоваться для написания кода.

Если мы хотим программировать для дела или для собственного удовольствия, нам нужен источник вдохновения в виде красивой архитектуры – хотя, возможно, работать с ней мы не сможем. Наша работа должна отражать красивую архитектуру, но при этом быть практичной. Самым чистым и красивым из всех интеллектуальных творений является мир чистой математики; мы можем многому научиться от него, но программировать в нем невозможно. Программа должна рабо-

тать, и здесь начинаются трудности. Иногда бывает легко заблудиться в методологии проектирования и забыть о том, что исходная цель была совсем другой. Кристофер Александер (Christopher Alexander), архитектурный отец паттернов проектирования, сказал следующее:

> На идее «методов проектирования» выросла целая академическая область, и меня называют одним из ведущих экспертов этих так называемых методов проектирования. Я очень сожалею о том, что это произошло, и хочу открыто заявить: я отвергаю самую идею методов проектирования как предмета изучения, потому что считаю, что изучение проектирования абсурдно отделять от практики. Люди, которые изучают методы проектирования без соответствующей практики, почти всегда оказываются несостоявшимися проектировщиками, у которых не осталось творческих сил, которые утратили потребность творить или никогда не имели ее. (Alexander, 1971)

Мы, программисты, должны создавать программы, которые работают, а не просто красиво смотрятся. Впрочем, эти две цели не всегда несовместимы. На рис. 14.6 показан мост Салгинатобель, спроектированный Робертом Майаром (Robert Maillart) и построенный в 1930 году. Швейцарец Майар изучал инженерное дело, но его работы и, особенно, мосты, являются примерами архитектурной красоты. Принципиально то, что они не просто красивы. Майар строил свои мосты, выигрывая контракты у конкурентов, и в ходе тендера на мост Салгинатобель он обошел предложения 19 других конкурентов. Строительство моста и дороги в то время обошлось всего около 700 000 швейцарских франков – менее 4 миллионов долларов в наши дни. Миниатюрным его никак не назовешь. Длина моста превышает 90 метров, из которых 80 метров проходят над ущельем ручья Салгина (Billington, 2000). Именно изящество и легкость структуры делают ее такой экономичной. Мост экономичен благодаря своей элегантности.

Возможно, главным достоинством Майара был его прагматизм. Он приходил к своим проектам по пути творческой интуиции. Он избегал любых украшений и имитации традиционных архитектурных стилей. Его строения не могли быть проанализированы математическими средствами того времени (при отсутствии компьютеров), поэтому доказать их надежность было невозможно. Майар оценивал жизнеспособность своих разработок при помощи упрощенного графического анализа. Если бы Майару пришлось дожидаться скрупулезной проверки своих проектов, ни один из них не был бы построен (он умер в 1940 году). Майар «обнаружил, что инновации, особенно в области строительства мостов, появляются не из лабораторий и математических теорий, а из офисов архитекторов и со строительных площадок. Числа играют важную роль в инженерном деле. И все же новаторский проект моста был

Рис. 14.6. *Мост Салгинатобель*

порожден визуально-геометрическим воображением, а не абстрактным числовым анализом или умозаключениями из общих теорий» (Billington, 1997, с. 1–2).

Программирование, как и архитектура, ориентируется на практический результат. Лучше избегать догм и сосредоточиться на том, что реально работает:

> Архитектура – опасная смесь всесилия и бессилия. Архитектор вроде бы занимается «творением мира», но чтобы его мысли были востребованы, он должен зависеть от прихотей других людей – клиентов, отдельных личностей или учреждений. Таким образом, в основе карьеры любого архитектора лежит непоследовательность, или, если говорить точнее, произвольность; архитектор сталкивается с произвольными наборами требований, с параметрами, установленными кем-то другим, со странами, о которых ему почти ничего неизвестно, с вопросами, о которых он имеет лишь отдаленное представление. Архитектор должен справляться с проблемами, которые оказались не под силу интеллектам, значительно превосходящим его собственный. Архитектура по определению является *хаотическим приключением*. (Koolhas et al., 1998, p. xix)

Архитектура является хаотическим приключением, потому что одной красивой архитектуры недостаточно. Основным законом как архитектуры, так и программирования является не только красота, но и практическая полезность.

Библиография

Abrahams, David, and Aleskey Gurtovoy. 2005. «*C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*». Boston, MA: Addison-Wesley.

Alexander, Christopher. 1971. «*Notes on the Synthesis of Form*», Preface to the paperback edition. Cambridge, MA: Harvard University Press.

Alpert, Sherman R., Kyle Brown, and Bobby Woolf. 1998. «*The Design Patterns Smalltalk Companion*». Boston, MA: Addison-Wesley.

Billington, David P. 2000. «The Revolutionary Bridges of Robert Maillart». *Scientific American*. July, pp. 85–91.

Billington, David P. 1997. «*Robert Maillart: Builder, Designer, and Artist*». New York, NY: Cambridge University Press.

Black, Andrew P., et al. 2007. «*Squeak By Example*». Square Bracket Publishing.

Bloch, Joshua. 2008. «*Effective Java*», Second Edition. Boston, MA: Addison-Wesley.[1]

Brand, Stewart. 1997. «*How Buildings Learn: What Happens After They're Built*», Revised Edition. London, UK: Phoenix Illustrated.

Calvino, Italo. 1986. «Why Read the Classics?» *The Uses of Literature*. Translated by Patrick Creagh. New York, NY: Harcourt Brace Jovanovich.

Conroy, Thomas J., and Eduardo Pelegri-Llopart. 1983. «An Assessment of Method-Lookup Caches for SmallTalk-80 Implementations». *Smalltalk-80: Bits of History, Words of Advice*. Ed. Glenn Krasner. Boston, MA: Addison-Wesley.

Curtis, William J. R. 1996. «*Modern Architecture Since 1900*», Third Edition. New York, NY: Phaidon Press.

De Botton, Alain. 2006. «*The Architecture of Happiness*». London, UK: Hamish Hamilton.

[1] Джошуа Блох «Java. Эффективное программирование» (перевод 1-го издания), Лори, 2002.

Gamma, Erich, et al. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley.[1]

Goldberg, Adele, and David Robson.1989. «*Smalltalk-80: The Language*». Boston, MA: Addison-Wesley.

Forman, Ira R., and Scott H. Danforth. 1999. «*Putting Metaclasses to Work: A New Dimension in Object Oriented Programming*». Boston, MA: Addison-Wesley.

Jencks, Charles. 2006. «*The New Paradigm in Architecture: The Language of Post-Modernism*». New Haven, CT: Yale University Press.

Josuttis, Nicolai M. 1999. «*The C++ Library: A Tutorial and Reference*». Boston, MA: Addison-Wesley.

Kaufmann, Edgar Jr. 1986. «*Fallingwater: A Frank Lloyd Wright Country House*». New York, NY: Abbeville Press.

Koolhas, Rem, et al. 1998. «*S, M, L, XL*», Second Edition. New York, NY: The Monacelli Press.

McCarthy, John. 1960. «Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I». *Communications of the ACM*, April 1960.

McConnell, Steve. 2004. Code Complete, Second Edition. Redmond, WA: Microsoft Press.[2]

Meyers, Scott. 2005. «*Effective C++ 55 Ways to Improve your Programs and Designs*». Boston, MA: Addison-Wesley.[3]

Norman, Donald. 1988. «*The Psychology of Everyday Things*». New York, NY: Basic Books.

Nuttgens, Patrick. 1997. «*The Story of Architecture*», Second Edition. New York, NY: Phaidon Press.

Petzold, Charles. 1999. «*Programming Windows*», Fifth Edition. Redmond, WA: Microsoft Press.

Rand, Paul. 1985. «*A Designer's Art*». New Haven, CT: Yale University Press.

---

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2007.

[2] Стив Макконнелл «Совершенный код. Практическое руководство по разработке программного обеспечения», Питер, 2009.

[3] Скотт Мэйерс «Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ», ДМК Пресс, 2006.

Simon, Herbert. 1996. «*The Sciences of the Artificial*». Cambridge, MA: MIT Press.

Stroustrup, Bjarne. 1985. «*The C++ Programming Language*». Boston, MA: Addison-Wesley.[1]

Stroustrup, Bjarne. 1994. «*The Design and Evolution of C++*». Boston, MA: Addison-Wesley.[2]

Sullivan, Louis H. 1896. «The tall office building artistically considered». *Lippincott's Magazine*, March 1896.

Thomas, David, et al. 2005. «*Programming Ruby: The Pragmatic Programmers Guide*», Second Edition. Raleigh, NC, and Dallas, TX: The Pragmatic Bookshelf.

Vandervoorde, David, and Nicolai M. Josuttis. 2002. «*C++ Templates: The Complete Guide*». Boston, MA: Addison-Wesley.

Wolfe, Tom. 1982. «*From Bauhaus to Our House*». London, UK: Jonathan Cape.

[1] Бьерн Страуструп «Язык программирования C++. Специальное издание», Бином, Невский Диалект, 2008.

[2] Бьерн Страуструп «Дизайн и эволюция языка C++», ДМК Пресс, Питер, 2006.

Послесловие

Строить красивее

Уильям Дж. Митчелл

Между программными системами и архитектурными строениями часто проводятся поверхностные аналогии. Однако между столь различными системами в действительности существует намного более сильное и формальное сходство, чем кажется на первый взгляд.

Код программных систем состоит из одномерных строк символов, взятых из четко определенных номенклатур и объединенных по точным синтаксическим правилам. При запуске на соответствующих машинах этот код должен давать полезные результаты.

Разумеется, архитектурные строения не одномерны, но в остальном у них имеется много общего с программным кодом. Они представляют собой трехмерные сборки дискретных физических компонентов, взятых из довольно четко определенных номенклатур компонентов, объединенных по относительно жестким правилам синтаксиса, и предназначенных для выполнения полезных целей. (На практике архитекторы более свободны в выборе номенклатур и синтаксиса, чем программисты.)

В обоих случаях «правила игры» могут определяться на уровне формальных грамматик. В общем случае формальная грамматика указывает, как именно должны объединяться компоненты. Говоря точнее, в соответствии со стандартным определением, используемым в лингвистике и информатике, формальная грамматика состоит из: конечного множества N нетерминальных символов; конечного множества T терминальных символов; конечного множества R правил замещения;

исходного состояния S. Правило замещения состоит из совокупности символов в левой части, стрелки в середине и еще одной совокупности в правой части. Оно указывает, что совокупность в левой части может заменяться совокупностью в правой части. Полная совокупность с правильной структурой создается рекурсивным применением правил замены R к исходному состоянию S. Язык, определяемый грамматикой, состоит из всех совокупностей терминальных символов, которые могут быть получены таким способом.

Формальные грамматики обычно применяются к номенклатурам слов и описывают процесс построения полных и правильных предложений, однако область их применения этим не ограничивается. Они также могут применяться к другим объектам с целью описания способов построения их полезных совокупностей.

В случае грамматики языка программирования объединяются распознаваемые символические имена, в правых и левых частях правил замещения стоят одномерные строки символов, и эти правила определяют полные и корректные выражения языка. В случае двумерной грамматики геометрических форм объединяются двумерные геометрические формы, в правых и левых частях правил замещения стоят двумерные совокупности форм, а грамматика определяет полный и корректный графический дизайн объекта. В случае трехмерных архитектурных грамматик объединяемыми нетерминальными объектами являются конструктивные линии и т. д.; терминальными объектами, объединяемыми в «скелетах» нетерминальных объектов, – фактические архитектурные компоненты, а правила замещения определяют корректные и полные композиции таких компонентов – иначе говоря, дизайны в архитектурном языке, определяемом грамматикой.

Несколько десятилетий назад мы с Джорджем Стини (George Stiny) опубликовали формальную грамматику знаменитых вилл великого итальянского архитектора эпохи Возрождения Андреа Палладио[1]. На ее основе строятся дизайны всех известных вилл Палладио, а также многочисленных убедительных подражаний (или, можно сказать, всех вилл, которые Палладио мог бы построить, если бы он прожил дольше и у него было больше клиентов). Также грамматика предоставляет убедительное объяснение базовых принципов архитектуры вилл Палладио. С тех пор были построены многочисленные архитектурные грамматики для других видов дизайна.

[1] George Stiny and William J. Mitchell «The Palladian Grammar», Environment and Planning B, vol. 5, no. 1, 5–18 (1978).

Одна из самых важных функций архитектурной грамматики заключается в отражении принципов модульности и иерархической организации, характеризующих работу в некотором определенном архитектурном стиле. В точно определенном и широко используемом языке классической архитектуры, например, колонна состоит из базы, ствола и капители. Капитель дополнительно делится на иерархию компонентов (разных для дорического, ионического и коринфского орденов) и т. д. Перемещаясь на более высокий уровень иерархии подсистем, мы видим, что расположенные с одинаковыми интервалами колонны образуют колоннады. Затем из колонн, архитравов и фронтонов образуются портики. Наконец, все компоненты и подсистемы в совокупности образуют завершенную, грамматически корректную, классическую композицию. Эти композиции, как и предложения, можно разделить на именованные составляющие.

С точки зрения геометрии или САПР компоненты и подсистемы здания являются дискретными формами, которые могут трансформироваться и собираться для создания более крупных пространственных композиций. С точки зрения канала поставок и строительства они являются материальными элементами, которые производятся (часто в длинной последовательности шагов, каждый из которых увеличивает полезность продукта), приобретаются, перевозятся в нужное место и монтируются в составе композиции.

С точки зрения эксплуатации построенного здания они являются заменяемыми компонентами. А с функциональной точки зрения они являются модулями, которые играют в здании определенную роль и могут объединяться с другими модулями для создания подсистем, выполняющих высокоуровневые задачи. При такой композиции модули не только вступают в пространственные отношения, радующие глаз, но и благодаря подобным отношениям могут передавать что-либо (например, структурную нагрузку) через свои интерфейсы.

Аналогичным образом в языках программирования предусмотрены средства разбиения кода на модули; иерархическое объединение модулей создает модули более высокого уровня, вплоть до завершенных программных систем. Как известно любому программисту, хороший код имеет четкую, логичную структуру модулей и иерархий, реализованную с использованием языковых средств абстракции и организационных конструкций. Организационная четкость классической архитектуры служит отличной моделью.

За редкими исключениями архитектурные строения также дополнительно подчиняются принципам внутреннего порядка. Например, колонны, выстроенные в линию, обычно разделяются равными промежут-

ками. Если вы хотите написать код, генерирующий модель колоннады в САПР, вам не придется задавать местонахождение каждой колонны по отдельности. Вы применяете итерацию, а параметр местонахождения автоматически увеличивается на каждом шаге. Другими словами, принципы архитектуры выражаются более элегантно и лаконично, давая читателю кода более содержательное представление о ней.

А если потребуется сгенерировать правильную сетку колонн? Для этого применяется вложенная итерация. Первая итерация формирует ряд колонн с равными интервалами, а затем вторая итерация повторяет этот ряд столько раз, сколько необходимо для создания сетки.

А если угловые колонны должны отличаться от внутренних? (Архитекторы часто выбирают такое решение из-за отличающихся структурных и других условий в углах здания.) Используйте условие: если текущая колонна находится в углу, она заменяется альтернативной структурой. Если вы хотите изменить расстояние между колоннами, чтобы подчеркнуть важность центральной оси, использовать разное оформление для внутренних и внешних колонн и т. д. – просто введите дополнительные условия.

Модульность, иерархия и периодичность ни в коей мере не исчерпывают всех принципов организации, применяемых в повседневной работе архитекторов. Внимательно проанализировав архитектурные компоненты, вы часто найдете закономерности в размерах и пропорциях, симметрии (и их искусное нарушение), вложенные самоповторения, как во фракталах, и параметрически изменяемые мотивы. Читатель без труда представит себе аналоги этих принципов в программном коде.

Но в некоторых ситуациях внутренний порядок вроде бы отсутствует. Скажем, архитектор может по какой-то причине разбросать колонны произвольным образом. Что тогда? Самым компактным способом описания подобных конфигураций является определение местоположения каждой отдельной колонны. Более компактного, более экономичного описания просто не существует.

Как показывают эти простые примеры, хороший архитектор не использует при проектировании метод «грубой силы», не ориентируется на один конкретный случай и, бесспорно, старается избегать неуклюжих «заплаток». В соответствии со сложными требованиями площадки и контекста, климата, выполняемых операций, цепочек поставки материалов и компонентов, строительных процессов и бюджетов, архитектор проектирует разнообразные и сложные здания. Но при этом он стремится руководствоваться концептуальной элегантностью – следуя принципам экономии средств и неукоснительно применяя свою версию «бритвы Оккама». Под внешним разнообразием и сложностью

красивых архитектурных произведений обычно обнаруживаются простые, элегантные принципы функциональной организации и формального порядка. Выявление этих принципов требует интеллектуальных усилий – что является критической частью восприятия и получения удовольствия от архитектуры.

Если вам удастся разобраться в этих принципах, вы сможете строить модели таких работ всего несколькими элегантными строками кода на стандартном языке программирования или (в менее традиционной среде) несколькими правилами замещения форм. Возможно, вам даже удастся обобщить эти принципы и написать программный код, который строит дизайны по заданным условиям и требованиям. Но если постигнуть эти принципы не удастся, то вам придется писать более длинный, менее творческий код.

Визуальная сложность здания обусловлена сложностью требований, на которые отвечает архитектор, и измеряется продолжительностью его «тупого» описания на уровне отдельных точек. Принципы, которыми руководствовался архитектор в своем ответе, обычно могут быть выражены в коротком фрагменте кода, выполнение которого генерирует всю эту сложность. Как правило, чем ниже соотношение длин короткого и длинного описания, тем красивее здание.

Получается, что архитекторы восхищаются красотой тех зданий, которые применяют несколько простых элегантных принципов для выполнения многих сложных задач. Аналогичным образом, программные архитекторы и программисты восхищаются красотой кода, который четко и лаконично (но без вреда для удобства чтения или сопровождения) решает много сложных задач. А ученые восхищаются красотой и выразительностью простых законов, описывающих разнообразные явления. Все это частные проявления общих принципов красоты.

Соавторы

Тиль Адам (Till Adam) провел молодость за изучением философии, сравнительной теории литературы, американской истории и музыковедения, зарабатывая на жизнь музыкой. Когда ему не удалось добиться богатства и знаменитости на этом поприще, он получил степень магистра наук в математике, информатике и экономике. За несколько лет участия в разработке свободно распространяемого ПО (прежде всего KDE) он научился программировать и был принят на работу в фирму Klarälvdalens Datakonsult AB, где он сейчас, среди прочего, координирует всю деятельность компании, связанную с KDE и Free Software. Тиль живет с женой и дочерью в Берлине, Германия.

Джим Блэнди (Jim Blandy) занимался сопровождением GNU Emacs для Free Software Foundation с 1990 по 1993 год, а также выпустил Emacs версии 19 с Ричардом Столменом. Он является одним из исходных проектировщиков системы контроля версий Subversion; также участвовал в разработке системы контроля версий CVS, GNU Debugger (GDB), библиотеке языковых расширений Guile и версии Emacs, созданной для редактирования генных цепочек. В настоящее время работает для Mozilla Corporation над SpiderMonkey – реализацией языка программирования JavaScript для Mozilla. Джим живет в Портленде, штат Орегон, с женой и двумя детьми.

Мирко Бём (Mirko Boehm) – разработчик KDE с 1997 года; участвовал в работе комитета KDE e.V. с 1999 по 2006 годы. Выпускник школы бизнеса Университета Хельмута Шмидта в Гамбурге, Германия. В личной жизни старается держаться подальше от компьютеров – читает книги, напечатанные на настоящей бумаге, и проводит время с семьей. В настоящее время работает в фирме Klarälvdalens Datakonsult AB в Берлине, Германия; специализируется на разработке кросс-платформенных и встроенных программ.

Кристофер Деннис (Christopher Dennis) – ведущий разработчик проекта JPC с момента его запуска в 2005 году. Крис перешел на Java во

время работы над докторской диссертацией в Оксфордском университете. До этого работал на разных языках, от машинного кода Z80 на клавиатурах с шестнадцатеричными кодами до PHP и JavaScript. Любитель запредельных ситуаций и хитроумных приемов программирования, Крис обожает писать компактный, элегантный код на любых языках программирования.

Дэйв Феттерман (Dave Fetterman) – технический директор Facebook, основатель проекта Facebook Platform. До поступления на работу в Facebook, в 2006 году, работал программистом над проектами Microsoft, включая .NET Common Language Runtime (CLR). Любит создавать программы для других разработчиков и подолгу рассказывать о них тем, кто согласится слушать. В 2003 году получил степень бакалавра в области прикладной математики и степень магистра в области компьютерных технологий в Гарвардском университете.

Кайр Фрейзер (Keir Fraser) – основатель XenSource (в настоящее время часть Citrix Systems), ведущий архитектор гипервизора Xen. Кайр реализовал исходную версию Xen в 2002 году во время учебы в аспирантуре в Компьютерной лаборатории Кембриджа; продолжал руководить разработкой по мере того, как Xen вырос в большой общественный проект. Получил кандидатскую степень за работу над безблокировочным механизмом управления параллельным выполнением в 2004 году и в том же году перешел в преподавательский состав.

Пит Гудлиф (Pete Goodliffe) – программист, журналист, лектор и автор, который никогда не остается на одном месте в цепи разработки и потребления программных продуктов. Популярная книга Пита «Code Craft» (No Starch Press)[1] представляет собой практическое и занимательное исследование ремесла программирования в целом. Книга занимает целых 600 страниц, дело не из легких! Пит обожает карри и не носит ботинки.

Георгиос Гусиос (Georgios Gousios) – ученый по профессии, программист по образованию, энтузиаст по жизни. В настоящее время работает над кандидатской диссертацией в Афинском университете экономики и бизнеса в Греции. В круг основных интересов входят программирование, качество программных продуктов, виртуальные машины и операционные системы; получил степень магистра наук с отличием в Манчестерском университете, Великобритания. Гусиос участвовал во многих проектах с открытым кодом, работал в различных исследовательских и рабочих проектах как в академической, так и в коммер-

[1] Пит Гудлиф «Ремесло программиста. Практика написания хорошего кода», Символ-Плюс, 2009.

ческой среде. Был руководителем проектов, ведущим архитектором и ведущим разработчиком в проекте SQO-OSS, где искал новые способы оценки качества программных продуктов. В академической среде Гусиос опубликовал 10 технических статей в авторитетных журналах. Является членом ACM, IEEE, Ассоциации Usenix и технического комитета Греции.

Дэйв Гроув (Dave Grove) – штатный ученый-исследователь Dynamic Optimization Group из исследовательского центра Т. Дж. Уотсона (фирма IBM). Основные интересы: анализ и оптимизация объектно-ориентированных языков, проектирование и реализация виртуальных машин, JIT-компиляция, оптимизация на основании оперативно полученных данных и уборка мусора. Вступил в проект Jalapeño в 1998 году, был одним из ключевых разработчиков исходной реализации оптимизирующего компилятора и системы адаптивной оптимизации. С тех пор, как проект Jalapeño был переведен на распространение с открытым кодом в виде Jikes RVM в 2001 году, является активным членом рабочей группы и руководящего комитета Jikes RVM.

Джон Клейн (John Klein) – старший технический специалист института SEI (Software Engineering Institute), где он работает над архитектурными методами построения систем и помогает отдельным лицам, группам и организациям повышать свою квалификацию в области программных архитектур. Перед поступлением в SEI Джон работал старшим архитектором в фирме Avaya, Inc., где занимался разработкой многорежимных агентов и архитектур коммуникационного анализа, а также созданием и усовершенствованием архитектуры Customer Interaction Software Product Line. Ранее Джон занимался проектированием программных архитектур в фирме Quintus; там он спроектировал первый коммерчески успешный многоканальный интегрированный проект ПО контакт-центра, а также руководил технологической интеграцией портфеля продуктов после того, как фирма Quintus купила две другие компании. Перед поступлением в Quintus Джон работал для разных компаний в области видеоконференций и сетевого видео. Его профессиональная карьера началась в Raytheon, где он разрабатывал программные и аппаратные решения для обработки радарных сигналов, мультиспектральной обработки изображений, параллельных архитектур и алгоритмов. Джон получил ученую степень бакалавра технических наук в Технологическом институте Стивенса, а также степень магистра технических наук в Северовосточном университете. Является членом ACM и Компьютерного общества IEEE.

Грег Лехи (Greg Lehey) провел свою долгую карьеру в Германии и Австралии; он работал в агентстве космических исследований Германии, а также в таких компаниях-производителях компьютерного обору-

дования, как Univac, Tandem, Siemens-Nixdorf и IBM, не говоря уже о работе для безымянных фирм-разработчиков в качестве «продвинутого» пользователя и для самого себя – в качестве консультанта. Ему приходилось заниматься разнообразными вещами: от разработки ядра до управления продуктами, от системного программирования до системного администрирования, от обработки спутниковых данных до программирования топливных насосов, от производства CD-ROM с портированными бесплатными программами до проектирования наборов команд DSP. Он был участником основной группы FreeBSD, президентом Австралийской группы пользователей UNIX. Занимался разработкой проектов FreeBSD и NetBSD, является автором книг «Porting Unix Software» и «The Complete FreeBSD, Fourth Edition» (обе книги выпущены издательством O'Reilly). Известно, что он также участвовал в разработке коммерческих программных продуктов. Грег ушел на пенсию в 2007 году и сейчас занимается поиском нового стиля жизни. Большую часть времени (которого по-прежнему не хватает) проводит за своими увлечениями: классической музыкой для деревянных духовых инструментов, кулинарией, пивоварением (разработал систему ферментации под контролем компьютера), садоводством, верховой ездой и фотографией. Интересуется рядом исторических тем, включая древние и малоизвестные европейские языки. Домашняя страница Грега находится по адресу *http://www.lemis.com/grog/*.

Панайотис Луридас (Panagiotis Louridas) начал заниматься компьютерами в 1980-х годах, еще в эпоху Sinclair ZX Spectrum; с тех пор он программирует на языке машин, и ему это нравится. Он получил диплом в области компьютерных технологий на факультете информатики Афинского университета, а также кандидатскую и докторскую степени в Манчестерском университете. За прошедшие годы занимался разработкой программных продуктов для частного сектора, а в настоящее время работает в GRNET (Greek Research and Education Network). Является членом исследовательской группы SENSE (Software Engineering and Security) при Афинском университете экономики и бизнеса, опубликовал ряд статей на различные темы, от антропологии до криптографии, от представления результатов измерений до программирования. Особенно увлекается поиском связей мира компьютерных технологий с другими областями.

Стивен Дж. Меллор (Stephen J. Mellor) – получивший международное признание специалист по созданию эффективных инженерных решений для разработки программного обеспечения. В 1985 году он вместе с Уордом опубликовал популярную трилогию «Structured Development for Real-Time Systems» (Prentice Hall), а в 1988 году были изданы его первые книги по объектно-ориентированному анализу. Стивен также

опубликовал книгу «Executable UML: A Foundation for Model-Driven Architecture» (Addison-Wesley Professional) в 2002 году. Его последняя книга, «MDA Distilled: Principles of Model-Driven Architecture» (Addison-Wesley Professional), вышла в 2004 году. Он активно участвует в работе Object Management Group; является председателем комитета, включившего исполняемые действия в UML, а недавно завершил работу над стандартом исполняемого UML. Его подпись стоит под Манифестом Agile. В течение двух сроков участвовал в работе Архитектурного комитета OMG, председательствовал в консультативном комитете по программированию IEEE, а до недавнего времени был главным научным консультантом отдела встроенных программ в Mentor Graphics.

Бертран Мейер (Bertrand Meyer) – профессор в области программирования Цюрихского федерального технологического института, ведущий архитектор фирмы Eiffel Software, в которой он руководил проектированием среды EiffelStudio и многочисленных библиотек. Является автором многих популярных книг, в том числе «Object-Oriented Software Construction» (Prentice Hall), лауреат премии Jolt Award. Также получил премии ACM Software System Award и Dahl-Nygaard Award за свои работы в области объектных технологий и Eiffel, почетный доктор Государственного политехнического университета Санкт-Петербурга. Основными темами его исследований являются объектные технологии, языки программирования и методы проверки программных продуктов. Активно занимается консультационной деятельностью и читает лекции.

Уильям Дж. Митчелл (William J. Mitchell) – профессор архитектуры, мультимедийного искусства и науки в Массачусетском технологическом институте (МТИ), руководитель группы Smart Cities в лабораториях MIT Media Laboratory и MIT Design Laboratory. Ранее был деканом школы архитектуры и планирования в МТИ. В число его последних работ входят книги «World's Greatest Architect» и «Imagining MIT».

Дерек Мюррей (Derek Murray) – аспирант Компьютерной лаборатории Кембриджского университета. Присоединился к проекту Xen в 2006 году, занимался усовершенствованием безопасности Xen на уровне переработки архитектуры управляющего стека. В настоящее время его работа направлена на улучшение отказоустойчивости в крупномасштабных распределенных системах, но время от времени он возвращается к низкоуровневому системному программированию. Дерек получил степень магистра наук в области высокопроизводительных вычислений в Эдинбургском университете в 2006 году и степень бакалавра наук в области информатики в Университете Глазго в 2005 году.

Риз Ньюман (Rhys Newman) стал поклонником Java во время работы над докторской диссертацией в Оксфордском университете более 10 лет назад, когда возраст языка Java составлял всего пару лет. В своих ранних исследованиях он продемонстрировал, что высокопроизводительная обработка видеоизображения в реальном времени возможна даже на ранних виртуальных машинах Java на базе JIT-компиляции, в «чистой» Java-среде. С тех пор он работал как в образовании, так и в промышленности, снова и снова доказывая, насколько гибкой, производительной и быстрой в действительности является платформа Java. За свою более чем 20-летнюю карьеру в области разработки ПО завоевал несколько отраслевых наград за техническое мастерство; недавно вернулся в Оксфорд для проведения революционных исследований в области решетчатых вычислений. JPC является одной из составляющих его нового исследования.

Майкл Найгард (Michael Nygard) старается упростить работу и поднять стандарты качества разработчиков по всей стране. Он готов разделить свою страсть и энергию к усовершенствованию с каждым встречным – иногда даже с разрешения последних. За лучшие годы своей 20-летней карьеры Майкл узнал, что значит быть профессиональным программистом, которому небезразличны понятия искусства и качества работы. Он всегда готов прийти на помощь коллегам, преданным своей работе, – «неравнодушным» разработчикам. С другой стороны, он не выносит апатии или впустую растраченных возможностей. Майкл проработал профессиональным программистом и архитектором почти 20 лет. За это время он создавал работоспособные системы для Правительства США, Министерства обороны, банковских, финансовых, сельскохозяйственных и торговых организаций. Как правило, Майкл отдавал себя без остатка создаваемым им системам. Опыт разработки в реальных условиях навсегда изменил его взгляды на программную архитектуру и разработку. Он сопровождал процесс рождения и «детский период» одного крупного сайта розничной торговли, а также обеспечивал «передвижную диагностику» для других видов интернет-коммерции. Благодаря этому у него сформировался особый взгляд на построение высокопроизводительных и надежных программных продуктов в условиях активно враждебной среды. Недавно Майкл опубликовал книгу «Release It! Design and Deploy Production-Ready Software» (Pragmatic Programmers), удостоенную премии Jolt Productivity Award в 2008 году. Другие его статьи можно найти по адресу *http://www.michaelnygard.com/blog*[1].

[1] Ряд статей Майкла Найгарда входит в сборник «97 этюдов для архитекторов программных систем», Символ-Плюс, 2010.

Иэн Роджерс (Ian Rogers) – научный сотрудник из исследовательской группы Advanced Processor Манчестерского университета. Его кандидатская исследовательская работа над двоичным транслятором Dynamite нашла коммерческое применение и сейчас входит во многие двоичные трансляторы, в том числе и в Rosetta фирмы Apple. Последние исследования Иэна относились к области проектирования языков программирования, сред времени выполнения и виртуальных машин – и прежде всего возможностям автоматического формирования и эффективного использования параллелизма. Иэн является ведущим соавтором и участником основной группы Jikes RVM.

Брайан Слеттен (Brian Sletten) – программист с гуманитарным образованием, специализирующийся на технологиях опережающего обучения. Работал системным архитектором, разработчиком, преподавателем и консультантом; выступал на конференциях во многих странах мира, пишет о веб-ориентированных технологиях для нескольких электронных журналов. Имеет практический опыт работы в оборонных, финансовых и коммерческих отраслях. Проектировал и строил сетевые матричные управляющие системы, интернет-игры, среды 3D-имитации/визуализации, распределенные вычислительные интернет-платформы, сети P2P и системы на базе Semantic Web. Получил степень бакалавра в области компьютерных технологий в Колледже Уильяма и Мэри; в настоящее время живет в Ферфаксе, штат Вирджиния. Является президентом Bosatsu Consulting, Inc. – профессиональной компании по обслуживанию, специализирующейся на веб-архитектурах, ресурсно-ориентированных вычислениях, Semantic Web, расширенных пользовательских интерфейсах, масштабируемых системах, консультациях в области безопасности и других технологиях конца XX – начала XXI века.

Диомидис Спинеллис (Diomidis Spinellis) – доцент факультета теории управления и технологии Афинского университета экономики и бизнеса в Греции. Основные направления исследований включают технологии разработки, компьютерную безопасность и языки программирования. Написал две книги из серии «Open Source Perspective», опубликованные издательством Addison-Wesley: «Code Reading» (премия Software Development Productivity Award в 2004 году) и «Code Quality» (премия Software Development Productivity Award в 2007 году). Также является автором десятков научных статей. Участник редакционного совета *IEEE Software*, автор периодической рубрики «Tools of the Trade». Диомидис участвовал в разработке FreeBSD, UMLGraph и других программных пакетов, библиотек и инструментов с открытым кодом. Он является обладателем степени магистра технических наук в области технологии программирования, а также Ph.D. в области компью-

терных технологий (обе степени получены в Имперском колледже Лондона). Диомидис входит в число старших членов ACM, а также является участником IEEE и Ассоциации Usenix.

Джим Уолдо (Jim Waldo) – выдающийся инженер из лаборатории Sun Microsystems, где он занимается исследованиями в области крупномасштабных распределенных систем следующего поколения. В настоящее время является техническим руководителем проекта Darkstar – многопоточной распределенной инфраструктуры для создания крупномасштабных многопользовательских сетевых игр и виртуальных миров. До поступления в Sun Labs был ведущим архитектором Jini – распределенной системы программирования на базе Java. Джим был редактором книги «The Evolution of C++ Language Design in the Marketplace of Ideas» (MIT Press), а также одним из авторов «The Jini Specification» (Addison-Wesley). Кроме того, он был сопредседателем Совета национальных академий, выпустившего книгу «Engaging Privacy and Information Technology in a Digital Age», которую он редактировал. Также Джим является свободным преподавателем факультета компьютерных технологий Гарвардского университета, где он преподает распределенную обработку данных и некоторые темы, находящиеся на стыке политики и технологии. Ph.D. в области философии Джим получил в Массачусетском университете (Амхерст).

Дэвид Вайсс (David Weiss) получил степень бакалавра математики в Юнион Колледже и степень магистра в области компьютерных технологий в Мэрилендском университете. В настоящее время возглавляет отдел исследований в области технологий программирования в Avaya Laboratories, занимаясь проблемой повышения эффективности программирования вообще и процессов разработки Avaya в частности. В последнем качестве он возглавляет ресурсный центр программных технологий Avaya. Ранее он был директором отдела исследований в области производства программного обеспечения Lucent Technologies в Bell Laboratories, проводившей исследования по повышению эффективности разработки. Перед поступлением в Bell Labs он руководил отделом повторного использования и сбора данных SPC (Software Productivity Consortium) – консорциума 14 крупных аэрокосмических компаний США. До поступления в SPC доктор Вайсс провел год в Бюро оценки технологий, где стал одним из соавторов оценочного анализа Стратегической оборонной инициативы. В 1985–1986 учебном году был приглашенным ученым в Институте Ванга и в течение многих лет занимался исследованиями в отделе компьютерных технологий и систем научно-исследовательской лаборатории ВМС (NRL) в Вашингтоне. Также работал программистом и математиком. Основные направления исследований Дэйва лежат в области технологии программирова-

ния, особенно в области процессов и методологий разработки, проектирования и измерения рабочих характеристик. Наибольшую известность получили изобретенная им методология «цель-вопрос-метрика», его работы по модульной структуре программных систем и его работа в качестве соавтора процесса Synthesis и его наследника, процесса FAST. Дэйв также является соавтором и соредактором двух книг: «Software Product-Line Engineering» и «Software Fundamentals: Collected Papers of David L. Parnas» (обе книги опубликованы издательством Addison-Wesley Professional).

Алфавитный указатель

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Г

Д

З

И

К

Л

М

H

O

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Я