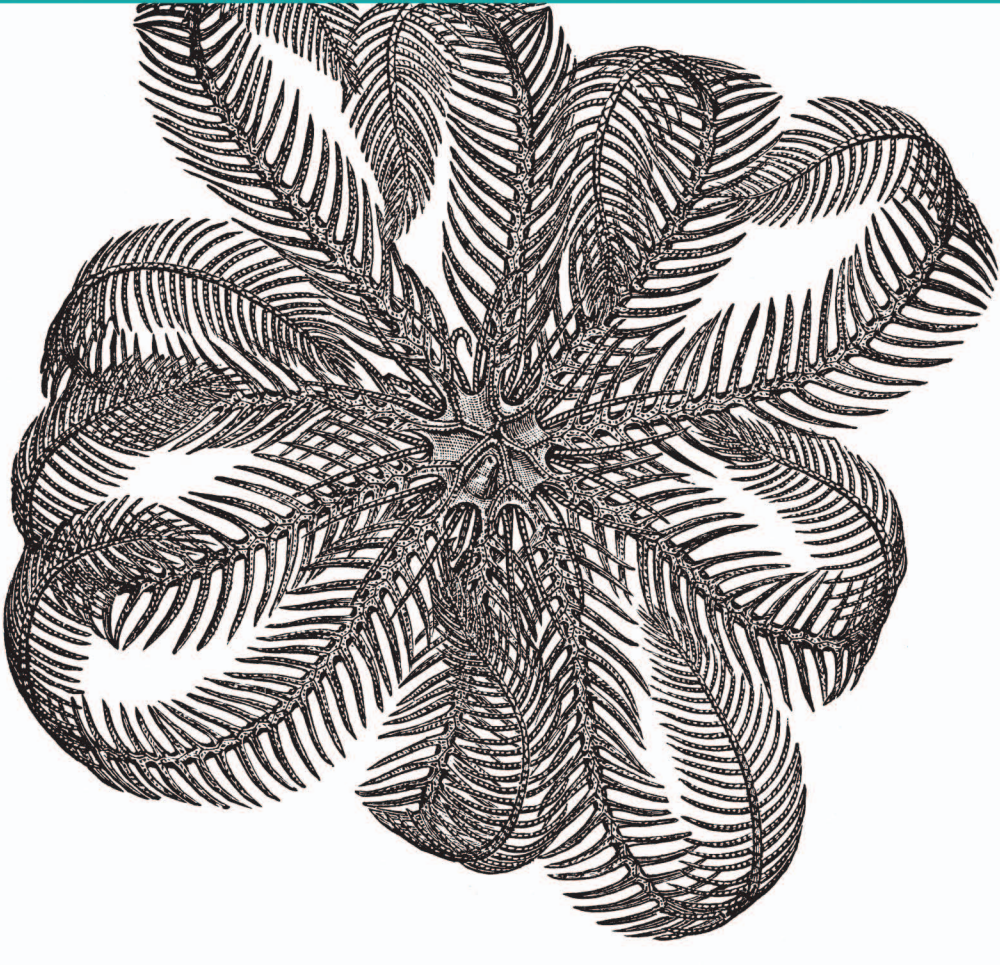
Приемы объектно-ориентированного программирования

ActionScript 3.0
Шаблоны проектирования



O'REILLY®

Adobe Developer Library

Уильям Сандерс
Чандима Кумаранатунг

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-189-9, название «ActionScript 3.0. Шаблоны проектирования» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

ActionScript 3.0
Design Patterns

William B. Sanders,
Chandima Cumaranatunge

O'REILLY®

ActionScript 3.0
Шаблоны проектирования

Уильям Сандерс, Чандима Кумаранатунг

Санкт-Петербург — Москва
2011

Уильям Сандерс, Чандима Кумаранатунг

ActionScript 3.0. Шаблоны проектирования

Перевод М. Нетова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*О. Меркулова*
Научный редактор	*С. Блощинский*
Корректор	*Д. Щепелева*
Верстка	*К. Чубаров*

Сандерс У., Кумаранатунг Ч.

ActionScript 3.0. Шаблоны проектирования. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 592 с., ил.

ISBN 978-5-93286-189-9

Теперь, когда язык ActionScript стал полноценным языком объектно-ориентированного программирования (ООП), часто используемые шаблоны проектирования являются идеальным средством решения многих повторяющихся задач во Flash- и Flex-приложениях. Использование шаблонов не только упрощает планирование и разработку сложных приложений, но и предоставляет решения для многих стандартных проблем, помогает в поддержке и развитии готовых приложений.

В данном издании представлены ключевые особенности ActionScript 3.0, основные ООП-концепции, такие как классы, абстрактность, наследование и полиморфизм, а также преимущества использования шаблонов проектирования. Затем детально рассматриваются конкретные шаблоны: Фабричный метод, Одиночка, Декоратор, Адаптер, Композиция, Команда, Наблюдатель, Стратегия, Состояние, Модель-Представление-Контроллер и Симметричный заместитель. Авторы приводят множество примеров различной степени сложности: веб-приложения для электронной коммерции, динамичные игры, запись и воспроизведение видео и многие другие.

Эта книга необходима любому разработчику Flash или Flex, желающему использовать продвинутые технологии ActionScript 3.0 в создании элегантных программных решений.

ISBN 978-5-93286-189-9
ISBN 978-0-596-52846-1 (англ)

© Издательство Символ-Плюс, 2010
Authorized translation of the English edition © 2007 O'Reilly Media Inc. This translation is published and sold by permission of O'Reilly Media Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

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 28.12.2010. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 37 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Билл посвящает эту книгу подрастающему поколению,
Рику и Оливии.

Чандима хотел бы посвятить данную книгу
своим родителям, Гайя и Лакшми.

Оглавление

Предисловие 11

Часть I. Постоянные изменения 23

Глава 1. ООП, шаблоны проектирования и язык ActionScript 3.0 25
Удовольствие от хорошей работы 25
Основы ООП 33
Абстрактность 34
Инкапсуляция 38
Наследование 48
Полиморфизм 60
Принципы разработки с помощью шаблонов проектирования 68
Использование интерфейсов, а не их имплементаций 72
Преимущество композиции 76
Планирование поддержки и развития приложений 85
План для вашего приложения: это не ваше дитя 88

Часть II. Порождающие шаблоны 93

Глава 2. Шаблон Фабричный метод 95
Что такое шаблон Фабричный метод? 95
Абстрактные классы в ActionScript 3.0 99
Минималистский пример 100
Сокрытие классов продуктов 105
Пример: печатная мастерская 106
Расширенный пример: цветная печать 111
Ключевые концепции ООП, использованные в шаблоне Фабричный метод 116
Пример: фабрика спрайтов 117
Пример: вертикальная стрелковая игра 122
Заключение 134

Глава 3. Шаблон Одиночка 135
Что такое шаблон Одиночка? 135
Ключевые концепции ООП, используемые в шаблоне Одиночка 137

Минималистский абстрактный Одиночка ... 140
Когда использовать шаблон Одиночка ... 148
Заключение ... 162

Часть III. Структурные шаблоны ... 165

Глава 4. Шаблон Декоратор ... 167
Что такое шаблон Декоратор? ... 167
Ключевые концепции ООП, используемые в шаблоне Декоратор ... 171
Минималистский абстрактный Декоратор ... 173
Применение простого шаблона Декоратор во Flash: картонная кукла ... 180
Декорация смертными грехами и райскими добродетелями ... 188
Динамический выбор компонентов и декораций: салон гибридных автомобилей ... 206
Заключение ... 218

Глава 5. Шаблон Адаптер ... 220
Что такое шаблон Адаптер? ... 220
Адаптеры объектов и адаптеры классов ... 222
Ключевые концепции ООП в шаблоне Адаптер ... 229
Пример: адаптер рулевого колеса ... 229
Расширенный пример: поворот машины с помощью мыши ... 238
Пример: адаптер для отображения списка ... 239
Расширенный пример: вывод списка новых книг издательства O'Reilly ... 245
Заключение ... 249

Глава 6. Шаблон Компоновщик ... 250
Что такое шаблон Компоновщик? ... 250
Ключевые особенности шаблона Компоновщик ... 253
Минималистский пример шаблона Компоновщик ... 254
Ключевые концепции ООП в шаблоне Компоновщик ... 265
Пример: списки мелодий ... 265
Пример: анимация составного объекта с помощью инверсной кинематики ... 270
Использование встроенной составной структуры Flash: список отображения ... 282
Заключение ... 293

Часть IV. Поведенческие шаблоны ... 295

Глава 7. Шаблон Команда ... 297
Что такое шаблон Команда? ... 297
Минималистский пример шаблона Команда ... 302
Ключевые концепции ООП в шаблоне Команда ... 306

Минималистский пример: Макрокоманды 306
Пример: манипулятор числами 310
Расширенный пример: разделяемые командные объекты 315
Расширенный пример: реализация отмены команд 317
Пример: подкаст радио 322
Расширенный пример: динамическое присвоение командного объекта 328
Заключение 334

Глава 8. Шаблон Наблюдатель 335
Что такое шаблон Наблюдатель? 335
Основные характеристики 337
Ключевые концепции ООП, используемые в шаблоне Наблюдатель 339
Минималистский абстрактный Наблюдатель 344
Пример: множество состояний и идентификация пользователей 349
Динамическое изменение состояний 358
Пример: различные отображения данных 374
Заключение 387

Глава 9. Шаблонный метод 388
Что такое Шаблонный метод? 388
Ключевые концепции ООП, используемые в Шаблонном методе 393
Минималистский пример: абстрактный Шаблонный метод 396
Использование гибкости Шаблонного метода 399
Выбор и воспроизведение звука и видео 403
Подцепляй 410
Заключение 416

Глава 10. Шаблон Состояние 417
Шаблон проектирования для создания автоматов 417
Ключевые концепции ООП, используемые в шаблоне Состояние 421
Минималистский абстрактный шаблон Состояние 422
Настоящий видеоплеер с состояниями 429
Расширение шаблона Состояние: добавление новых состояний .. 436
Добавление новых состояний и возможности записи 444
Заключение 459

Глава 11. Шаблон Стратегия 461
Что такое шаблон Стратегия? 461
Ключевые характеристики 462
Модель шаблона Стратегия 462
Ключевые концепции ООП, используемые в шаблоне Стратегия 463

Минималистский абстрактный шаблон Стратегия 466
Больше конкретных стратегий и контекстов 470
Работа со строковыми стратегиями 479
Заключение 488

Часть V. Множественные шаблоны 491

Глава 12. Шаблон Модель–Представление–Контроллер 493
Что такое шаблон Модель-Представление-Контроллер (МПК)? 493
Взаимодействие между элементами МПК 495
Внутренние шаблоны МПК 497
Минималистский пример шаблона МПК 498
Ключевые особенности шаблона МПК 510
Ключевые концепции ООП в шаблоне МПК 510
Пример: карты погоды 511
Расширенный пример: инфракрасные карты погоды 519
Пример: машинки 526
Специальные представления 532
Добавление преследующей машинки 535
Заключение 537

Глава 13. Шаблон Симметричный заместитель 539
Одновременные ходы и результаты в играх 540
Шаблон Симметричный заместитель 543
Ключевые концепции ООП
в шаблоне Симметричный заместитель 546
Интерфейс игрока 549
Судья 550
Информация, разделяемая через Интернет 555
Классы игрока и заместителя 558
Служебные классы и файл документа 566
Заключение 570

Алфавитный указатель 572

Предисловие

После того как ActionScript эволюционировал от нескольких операторов для Flash до полноценного языка программирования веб-приложений (в своих последних версиях), у нас появилась возможность использовать в своих разработках на нем самые современные технологии, созданные для объектно-ориентированного программирования (ООП). Появление ActionScript 3.0 ознаменовало начало новой эры в программировании Flash и Flex, поскольку теперь он соответствует стандарту ECMAScript для языков разработки в Интернет. Многие возможности таких языков, как C++ и Java, доступны теперь и в ActionScript 3.0.

Вместе с новыми возможностями в языке ActionScript 3.0 к нам пришли современные способы разработки, изменился даже сам процесс *мышления* на данном языке. Хотя мы предполагаем, что большинство читателей этой книги в некоторой степени знакомы с ООП, но, подобно переходу от последовательного или процедурного программирования к ООП, переход к программированию с использованием шаблонов проектирования является скачком вверх для ООП-программистов. Мы думаем, что поскольку в ActionScript 3.0 появились возможности разрабатывать более сложные программные структуры, программистам Flash и Flex потребуются соответствующие техники программирования для успешной работы с ними.

Владея процессом программирования с использованием шаблонов проектирования[1], вы можете писать более эффективный ООП-код и многократно использовать его в других ваших программах. Разработчики, умеющие работать в больших командах и разбирающиеся в современных структурах, для оперирования с которыми были созданы шаблоны проектирования и ООП, занимают более высокооплачиваемые позиции. Использование шаблонов проектирования не только повышает ваше мастерство разработки сложных приложений, но и облегчает само программирование. Большинство трудностей при создании крупных сложных приложений возникает из-за их плохого планирования и неудобных конструкций. Шаблоны проектирования не только предоставляют решения для многих стандартных проблем, но и помогают в процессах обслуживания и изменения готовых программ. Словарь шаблонов проектирования также очень важен, поскольку его освоение позволит вам

[1] В разговорной речи обычно говорится просто «шаблон» или «паттерн». – *Прим. перев.*

стать членом современного сообщества разработчиков, свободно разговаривающего на этом языке.

Для кого эта книга

Мы планировали написать книгу для пользователей ActionScript 3.0 с уровнем подготовки от среднего до продвинутого. В отличие от разработчиков на некоторых других языках программирования, таких как Java, которые обычно являются выпускниками компьютерных факультетов, большинство пользователей ActionScript 3.0 изучили его в процессе работы во Flash. В результате уровень их теоретических знаний варьируется значительно сильнее, как и их общий практический опыт программирования. Мы, конечно, понимаем, что некоторые программисты на ActionScript 3.0 имеют профессиональное высшее образование, и для них большая часть вводного материала из первой главы будет излишней. Но мы также уверены, что часть читателей с уровнем подготовки пониже будет изучать объектно-ориентированное программирование одновременно с освоением стандартных шаблонов проектирования, и у них, возможно, меньше практики программирования.

При таком разбросе исходных позиций обязательно будет что-то слишком сложное для одних и чересчур простое для других. Однако главная задача этой книги – объяснить, как следует использовать различные шаблоны проектирования. Наша целевая аудитория – разработчики ActionScript со средним уровнем подготовки. Сначала мы приводим весь материал, необходимый им, чтобы стать продвинутыми. А уже для продвинутого пользователя мы предоставляем объяснения и примеры применения шаблонов проектирования в ActionScript 3.0.

Как организована данная книга

Структура данного издания во многом повторяет структуру книги «Design Patterns: Elements of Reusable Object-Oriented Software», написанной Эриком Гаммой (Erich Gamma), Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides) (издательство Addison-Wesley, 1995)[1].

Первая часть данной книги содержит только главу 1, которая является вводной для темы шаблонов проектирования. Мы добавили ее для читателей с минимальным уровнем знания объектно-ориентированного программирования. Более опытным пользователям, вероятно, можно

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2007.

пропустить этот краткий обзор ООП и сразу перейти к материалу, посвященному шаблонам проектирования.

Часть I «Постоянные изменения».

Глава 1 «Объектно-ориентированное программирование, шаблоны проектирования и ActionScript 3.0».

Части II, III и IV – главные в этой книге. В них рассматриваются базовые шаблоны проектирования, сгруппированные по категориям: шаблоны *порождающие, структурные* и *поведенческие*. Примеры таких шаблонов представлены в каждой части, но мы не включали в них абсолютно все шаблоны из книги Э. Гаммы и его коллег.

Часть II «Порождающие шаблоны».

Глава 2 «Шаблон Фабричный метод».

Глава 3 «Шаблон Одиночка».

Часть III «Структурные шаблоны».

Глава 4 «Шаблон Декоратор».

Глава 5 «Шаблон Адаптер».

Глава 6 «Шаблон Компоновщик».

Часть IV «Поведенческие шаблоны».

Глава 7 «Шаблон Команда».

Глава 8 «Шаблон Наблюдатель».

Глава 9 «Шаблонный метод».

Глава 10 «Шаблон Состояние».

Глава 11 «Шаблон Стратегия».

Часть V состоит из двух глав, посвященных использованию множественных шаблонов при разработке программ. Конструкции типа Модель-Представление-Контроллер и Симметричный заместитель включают в себя более одного шаблона проектирования. Эти главы организованы подобно прочим в плане объяснения работы сложных шаблонов. Однако многовариантные конструкции основываются больше на диаграммах объектов, чем на диаграммах классов.

Часть V «Множественные шаблоны».

Глава 12 «Шаблон Модель-Представление-Контроллер».

Глава 13 «Шаблон Симметричный заместитель».

Каждая глава, посвященная шаблонам проектирования, организована так, чтобы обеспечить максимально ясное понимание назначения и способа использования данного шаблона. Следующие разделы, хотя и не обязательно в таком порядке, присутствуют в каждой главе о шаблонах проектирования:

- Что это за шаблон?
- Основные характеристики и возможности данного шаблона.
- Формальная модель шаблона, включая его диаграмму классов.
- Ключевые концепции ООП, использованные в данном шаблоне.
- Небольшой абстрактный пример применения шаблона.
- Прикладные примеры его использования.

Мы построили нашу книгу таким образом для того, чтобы представить нашим читателям всестороннюю картину для каждого шаблона проектирования. Обсуждая предназначение шаблона и его свойства, мы уделяем основное внимание его функциям и структуре. Формальная модель и диаграмма классов дают нам возможность шире взглянуть на данный шаблон, вы познакомитесь с его структурой и связанными с ним классами и интерфейсами. Мы включили также рассмотрение некоторых ключевых концепций ООП для различных шаблонов. Сделали мы это по двум причинам. Во-первых, пользователи среднего уровня смогут лучше усвоить эти концепции на практике, а значит, и глубже понять ООП в целом. Во-вторых, мы рассчитываем, что продвинутые пользователи смогут использовать эти концепции как условные обозначения для быстрого определения того, как данный шаблон проектирования структурирован.

Что необходимо для работы с книгой

Для изучения приводимых примеров программ вам потребуется либо Flash CS3, либо Flex 2. Все приложения были разработаны с помощью Flash IDE, поэтому пользователям Flex 2 потребуется вносить в них некоторые изменения, особенно в те места, где для создания возможностей использовались инструменты и компоненты рисования Flash.

Несколько примеров используют Flash Media Server 2 (FMS2). Такие примеры могут быть созданы и опробованы с помощью версии FMS2 для разработчиков, доступной для бесплатной загрузки по адресу *http://www.adobe.com/downloads/*. Вам потребуется либо ОС Windows, либо ОС Linux для запуска FMS2. Если у вас Макинтош типа «Mactel», вы можете запустить на нем ОС Windows, но если у вас более старая версия Макинтоша, работающая на процессоре Motorolla, вам потребуется локальный или интернет-доступ к Windows или Linux-серверу с FMS2. Либо вы можете просто пропустить примеры с FMS2.

Повторение – мать учения

Читая эту книгу, вы заметите, что материал в ней иногда повторяется. Мы знаем, что у разных людей стиль обучения различен. Одним достаточно общей концепции, вторые учатся на примерах, третьи – на мета-

формах или комбинациях вышеперечисленного. Плюс к этому, возможно, существуют и другие методики, о которых мы не слышали. В книге мы объясняем одни и те же вещи несколькими различными способами в надежде на то, что если вы не поймете материал в первом варианте, то уж точно поймете во втором.

Кроме того, мы полагаем, что обсуждение одних и тех же идей и концепций по-разному и в различных контекстах формирует лучшее *сущностное* понимание этих идей и концепций. Изучив немалый материал по теме шаблонов проектирования, включая опубликованные книги, статьи и информацию в Интернете, мы обнаружили, что некоторые детали описаны там недостаточно аккуратно. Мы очень старались избежать в нашей книге ошибок; рассмотрение понятий в различных контекстах дало нам возможность более четко формулировать то, что мы имеем в виду. Решающее слово неизменно останется за книгой «Design Patterns: Elements of Reusable Object-Oriented Software», поэтому если у вас останутся какие-либо вопросы по нашим разъяснениям, вы всегда сможете уточнить их в указанном оригинале.

В течение нескольких последних лет появился ряд статей, книг, диссертаций и исследований, предлагающих дальнейшее развитие изначальных шаблонов проектирования. Некоторые из этих документов были весьма полезны и даже поддержаны членами «банды четырех» (GoF) – Эриком Гаммой (Erich Gamma), Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides). Другие оказались менее удачными, особенно в смысле легкости изучения по ним шаблонов проектирования, и имели тенденцию еще больше усложнять и без того непростую тему. Исходя из этого мы не стали сильно отклоняться от пути, указанного нам в исходном тексте «банды четырех».

Руководство пользователя

Эта книга по своей сути является введением в относительно сложную тему написания повторно используемого ООП-кода на ActionScript 3.0. Как и словосочетание «гигантская букашка», фраза «элементарное введение в сложную тему» является оксюмороном. Опытные разработчики, вероятно, захотят, чтобы в ней было меньше элементарного, а менее продвинутые программисты могут потребовать, чтобы в ней было больше подготовительного материала.

Поскольку мы не можем определить уровень подготовки каждого нашего читателя, мы убедительно просим вас прочитать оглавление, пролистать заинтересовавшие вас главы и найти то, что нужно именно вам. Определите свой текущий уровень и используйте книгу в соответствии с ним. Кто-то прочтет ее от корки до корки, а для кого-то данная книга послужит только справочным материалом для поиска того, как

ActionScript 3.0 работает с различными вариантами шаблонов проектирования. В конце концов, это ваша книга, и вы должны использовать ее с максимальной для себя пользой.

Использование Flex 2

Мы разрабатывали все наши примеры с использованием Flash CS3. Так что если вы захотите использовать их во Flex 2, вам потребуется их несколько изменить. В некоторых случаях примеры были созданы с применением пользовательских классов на основе MovieClip, разработанных в IDE Flash CS3 и сохраненных в Библиотеке. Такие примеры невозможно напрямую повторить во Flex 2, поэтому вам придется строить обходные пути.

Использование Flash Media Server 2

В книге есть несколько примеров, использующих Flash Media Server 2 (FMS2). Для их работы требуется только версия FMS2 для разработчиков, которую можно бесплатно загрузить с сайта Adobe по адресу: *http://www.adobe.com/devnet/flashmediaserver/*. Либо вы можете пропустить эти примеры или использовать другое программное обеспечение.

Что еще вам потребуется

В идеальном мире все читатели нашей книги прекрасно знали бы тему объектно-ориентированного программирования и язык ActionScript 3.0 заранее. Однако ActionScript 3.0 появился (не в бета-версии) во Flex 2 всего за шесть месяцев до выхода оригинала этой книги, а во Flash – примерно во время ее публикации. Так что есть вероятность, что вы совсем не знакомы с ActionScript 3.0, а данная книга вовсе не является введением в этот язык. Тогда как минимум вам следует держать под рукой «ActionScript 3.0 Reference Guide», а также прочую документацию по нему, идущую вместе с Flash CS3.

Мы настоятельно рекомендуем вам приобрести книгу «Design Patterns: Elements of Reusable Object-Oriented Software». По крайней мере, возьмите ее на время в библиотеке. Еще одна бесценная, на наш взгляд, книга – это чрезвычайно легкая и понятно написанная «Head First Design Patterns» Эрика и Элизабет Фриман (Eric Freeman, Elisabeth Freeman) (издательство O'Reilly, 2004). Хотя там все примеры написаны на Java, из нее можно узнать очень многое о шаблонах проектирования и ООП. (Перевод Java-примеров на язык ActionScript 3.0 позволит вам отлично изучить его, но перед этим занятием обязательно просмотрите примеры шаблонов проектирования в нашей книге.)

Если у вас еще нет хорошей книги по ActionScript 3.0, советуем приобрести «ActionScript 3.0 Cookbook» Джои Лотта (Joey Lott), Деррона Шалла (Darron Schall) и Кейт Питерс (Keith Peters) (издатель-

ство O'Reilly, 2006)[1] и «Essential ActionScript 3.0» Колина Мука (Colin Moock) (издательство O'Reilly, 2007)[2]. Для очень краткого введения в ActionScript 3.0 прочитайте сокращенное издание «ActionScript 3.0 Programming: Overview, Getting Started, and Examples of New Concepts» Билла Сандерса (Bill Sanders) (издательство O'Reilly, 2007). Еще вам потребуется книга по ООП на ActionScript 3.0. Мы включили введение в ООП в первую главу нашей книги, его будет пока достаточно; книги К. Мука и Дж. Лотта также содержат много материала по этой теме. Однако книга, посвященная основным концепциям ООП, все-таки должна присутствовать в вашей библиотеке, если вы собираетесь писать программы на уровне шаблонов проектирования.

Соглашения, принятые в этой книге

В этой книге используются следующие типографические обозначения:

Рубленый шрифт

Применяется для выделения интерфейса – меню, кнопки, быстрые клавиши (такие как Alt и Ctrl).

Жирный шрифт

Указывает на текст, который должен быть введен пользователем.

Курсив

Указывает на новые термины, веб-ссылки (URL), адреса электронной почты, имена файлов и их расширения, пути поиска файлов, имена каталогов и утилиты ОС UNIX.

`Моноширинный шрифт`

Указывает на команды, опции, переключатели, переменные, атрибуты, ключи, функции, типы, классы, пространства имен, методы, модули, свойства, параметры, значения, объекты, события, обработчики событий, XML-метки, HTML-метки, а также на результаты работы команд.

`Жирный моноширинный шрифт`

Обозначает текст, который должен быть введен пользователем дословно.

`Моноширинный курсив`

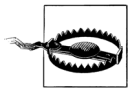
Обозначает текст, который должен быть заменен значениями, определяемыми пользователем.

---

[1] Дж. Лотт, Д. Шалл, К. Питерс «ActionScript 3.0. Сборник рецептов». – Пер. с англ. – СПб.: Символ-Плюс, 2007.

[2] К. Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

Таким способом выделяются советы, предложения и примечания общего характера.



Таким способом выделяются предупреждения и предостережения.

Использование примеров программ

Эта книга призвана помочь вам в работе. В общем вы можете использовать приведенный в ней код в собственных программах или в создаваемой вами документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, не требуется разрешение, чтобы включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске нужно получить разрешение. Можно без ограничений цитировать книгу и примеры в ответах на вопросы. Но чтобы включить значительные объемы кода в документацию по собственному продукту, нужно получить разрешение.

Мы будем благодарны вам за ссылку на данную книгу при цитировании, хотя и не требуем этого. В ссылке обычно приводятся название книги, имена авторов, издательство и ISBN, например: «ActionScript 3.0 Design Patterns by Bill Sanders and Chandima Cumaranatunge. Copyright 2007 O'Reilly Media, Inc., ISBN 978-0-596-52846-1».

Если вы полагаете, что планируемое вами использование кода выходит за рамки законного или разрешений, оговоренных выше, пожалуйста, обращайтесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Пожалуйста, присылайте все комментарии и вопросы, касающиеся данной книги, в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

На веб-странице данной книги содержатся найденные в ней ошибки, все ее примеры и дополнительная информация. Адрес этой страницы:

http://www.oreilly.com/catalog/9780596528461

Все комментарии и вопросы по данной книге присылайте по адресу:

bookquestions@oreilly.com

Чтобы узнать больше информации о наших книгах, конференциях, центре ресурсов и сети издательства O'Reilly, посетите наш веб-сайт:

http://www.oreilly.com

Safari® Enabled

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

У Safari есть преимущество перед обычными электронными книгами. Это виртуальная библиотека, которая позволяет легко находить тысячи лучших технических изданий, копировать примеры программ, загружать отдельные главы и быстро получать точную и актуальную информацию. Библиотека бесплатна и расположена по адресу *http://safari.oreilly.com*.

Благодарности

С тех пор как благодаря аЙо Бинити (aYo Binitie) мы познакомились с шаблонами проектирования, они стали предметом нашего пристального интереса. Как и многие разработчики на ActionScript, мы благодарны Колину Муку (Colin Moock) за начальное освещение темы шаблонов проектирования на ActionScript в книге «Essential ActionScript 2.0»[1]. Мы благодарны также Эрику Фриману (Eric Freeman) и Элизабет Фриман (Elisabeth Freeman) за их замечательную книгу «Head First Design Patterns»; хотя все примеры в ней на Java, она помогла нам оценить применение шаблонов проектирования в ActionScript 3.0. И как основателям темы шаблонов проектирования мы должны выразить нашу признательность «банде четырех», написавшей «Design Patterns: Elements of Reusable Object-Oriented Software», – Эрику Гамме (Erich Gamma), Ричарду Хелму (Richard Helm), Ральфу Джонсону (Ralph Johnson) и Джону Влиссидесу (John Vlissides). Мы провели немало времени, сосредоточенно изучая эту книгу.

Несколько людей из компании Adobe были так добры, что уделили нам свое время, помогли с ActionScript 3.0 и дали нам несколько идей насчет шаблонов проектирования. Вот они: Крис Нууйа (Chris Nuuja), Эрика Нортон (Erica Norton), Джеффри Уильямс (Geoffrey Williams), Грант Скиннер (Grant Skinner), Джеффри Мотт (Jeffrey Mott), Майк Дауни (Mike Downey), Нивеш Райбхандари (Nivesh Rajbhandari), Питер

[1] К. Мук «ActionScript 2.0. Основы». – Пер. с англ. – СПб.: Символ-Плюс, 2006.

Дехаан (Peter DeHaan), Роберт Пеннер (Robert Penner), Гари Гроссман (Gary Grossman), Али Миллс (Ali Mills), Френсис Ченг (Francis Cheng), Дэйвид Менделс (David Mendels), Гордон Смит (Gordon Smith), Роджер Гонсалез (Roger Gonzalez), Шо Кувамото (Sho Kuwamoto), Фрэнсис Чен (Francis Chen), Эмми Хуанг (Emmy Huang), Вернер Шарп (Werner Sharp), Джоан Тан (Joan Tan), Фил Коста (Phil Costa), Молли Гардинер (Mally Gardiner), Аса Вилок (Asa Whillock), Крис Хок (Chris Hock), Тарек Альджабер (Tareq Aljaber), Сан Хонг (San Khong) и Питер фон дем Хаген (Peter von dem Hagen).

Некоторые разработчики из Flash-сообщества помогли нам глубже понять тему использования шаблонов проектирования в ActionScript 3.0. Это Питер Холл (Peter Hall), Арал Балкан (Aral Balkan), Роберт Пеннер (Robert Penner), Бо Амбер (Beau Ambur), Стефан Рихтер (Stefan Richter), Джои Лотт (Joey Lott), Гай Ватсон (Guy Watson), Кейт Питерс (Keith Peters), Уил Лоу (Will Law) и Брайан Лессер (Brian Lesser). Джонатан Кей (Jonathan Kaye), реализовавший абстрактный автомат на языке ActionScript, посодействовал нам с шаблоном «Состояние», который послужил моделью для реализации в данной книге.

Мы очень благодарны также Марго Малей Хатчисон (Margot Maley Hutchison) из компании Waterside Productions за помощь при заключении контракта с издательством O'Reilly. Как и всегда, Марго удалось сильно упростить этот процесс.

Мы благодарны профессору Джону Грею (John Gray), главе программы «Мультимедийный веб-дизайн и разработка» (Multimedia Web Design and Development, MVDD) в университете Хартфорда. Его энтузиазм и поддержка по всем вопросам, касающимся Интернета и веб-разработок, исследований и обучения, обеспечили нам богатую идеями среду и множество удивительных открытий на нашем пути в неизменно расширяющуюся вселенную исследуемых нами технологий.

Технические консультанты

Наши технические консультанты сделали для нас гораздо больше, чем им полагалось. Некоторые из них были экспертами по шаблонам проектирования на C# и Java, но были не знакомы с ActionScript. К счастью, ActionScript 3.0 выглядит и действует подобно прочим языкам ООП, поэтому они смогли нам помочь. Главной в этой группе была Адриана Декер (Adrienne Decker), преподаватель департамента компьютерных и инженерных наук государственного университета Нью-Йорка (SUNY), г. Буффало. После посещения ее выступления на конференции по объектно-ориентированному программированию, системам, языкам и приложениям (OOPSLA) в 2006 г. в Портленде, штат Орегон, стало ясно, что Адриана не только очень много знает о шаблонах проектирования, но и умеет просто и доходчиво рассказывать об их внутренней работе. Она нам очень помогла. В этом проекте участвовали еще

двое ученых: доктор Джеймс Хейлотис (James Heilotis) и доктор Аксель Шрейнер (Axel Schreiner), оба из Рочестерского института технологий, департамент компьютерных наук. Профессор Шрейнер представил на конференции OOPSLA 2006 доклад об их шаблоне проектирования Симметричный заместитель. Как показано в их статье «A Pattern for Distributing Turn-Based Games», новый шаблон оказался отличным инструментом для создания интернет-игр, использующих Flash и Flash Media Server 2. Мы полагаем, что, учитывая новизну и необычность шаблона Симметричный заместитель, было совершенно обоснованно попросить их о технической рецензии на наше объяснение и реализацию этого шаблона. Они нам очень помогли, проследив за верным изложением нами их идей, и мы им обоим очень благодарны.

Нам повезло работать с Тоддом Андерсеном (Todd Anderson). Тодд отлично разбирается в ActionScript 3.0 и шаблонах проектирования. Он чрезвычайно помог нам, и мы особенно благодарны ему за его острый взгляд и точные комментарии. Даррен Ричардсон (Darren Richardson) был нашим техническим консультантом с самого начала нашей работы. Он также привнес в наши работы рассмотрение некоторых аспектов интернационализации, не дав нам стать слишком этноцентричными.

У нас была также уникальная возможность поработать с сообществом Flash-программистов Нью-Йорка (FCNY), возглавляемым Джином-Чарльзом Карелли (Jean-Charles Carelli). Проявив большой энтузиазм, члены этой группы неоднократно просматривали и редактировали наши рукописи в качестве дополнительных технических консультантов. Их комментарии были очень полезны для нас и отражали интересный срез их опыта как разработчиков-практиков. Среди тех, кто особенно способствовал этому процессу, были Тайлер Ларсон (Tyler Larson), Джим Кременс (Jim Kremens), Доминик Танкреди (Dominic Tancredi), Шари Холтер (Shari Halter), Джеймс О'Рейли (James O'Reilly), Эндрю Хант (Andrew Hunt), Брайан Вейсентал (Brian Weisenthal), Оскар Трелес (Oscar Trelles), Сет Хилинджер (Seth Hillinger), Лиза Ларсон (Lisa Larson) и Эдвард Скрипа (Edward Skrypa).

Редакторы

Нашему ведущему редактору, Робин Томас (Robyn Thomas), приходилось не только выделять смысл из наших высокотехнологичных текстов, но и направлять нас в правильном использовании английского языка, гарантирующем, что мы говорим именно то, что подразумеваем. Она управляла также множеством технических редакторов, работающих с главами нашей книги, собирала все эти кусочки вместе, расставляя их по своим местам, и вообще делала процесс написания книги настолько приятным, насколько это только возможно. Стив Вейс (Steve Weiss), наш издатель, всячески нас поддерживал и следил за общей организацией всего процесса. Стив всегда открыт новым перспективам

и всему креативному; скорее всего потому, что он и сам очень творческая личность.

Авторы

Билл Сандерс (Bill Sanders), PhD, и Чандима Кумаранатунг (Chandima Cumaranatunge), PhD, являются профессорами программы мультимедийного веб-дизайна и проектирования (MWDD) в университете Хартфорда. Билл ведет курсы по Flash, ActionScript, Flash Media Server 2, PHP, C#, SQL, CSS, XHTML и другим языкам для разработки веб-приложений. Он опубликовал 44 книги по компьютерной тематике, создавал программное обеспечение на языках от Basic до Flash Media Server ActionScript и работал консультантом в различных программистских компаниях. Чандима читает введение в MWDD, в основном посвященное Flash и ActionScript, курс по играм на Flash и ActionScript и курс по образовательным технологиям в медицинском колледже. Недавно он получил грант на преподавание экспериментального курса по робототехнике.

Билл Сандерс

Билл хотел бы поблагодарить своего соавтора, Чандима, за приятное и плодотворное сотрудничество в написании данной книги и за то, что с ним всегда можно было профессионально обсудить тему шаблонов проектирования. Билл хотел бы также поблагодарить свою жену Делию за то, что, пока она заканчивала свою докторскую диссертацию, ей приходилось терпеть наше сосуществование в режиме двух вечно занятых текстовых процессоров. Наш маниакально навязчивый большой швейцарский зенненхунд Вилде возвращал нам обоим чувство реальности; он всегда знал, что действительно важно – сходить погулять.

Чандима Кумаранатунг

Чандима считает, что ему очень повезло: он нашел в Билле не только партнера по написанию книги, но и профессионального наставника и настоящего друга. Он очень благодарен также своей жене Решмааль за ее всемерную поддержку и терпение по отношению к его долгой писательской работе вдали от семьи. И наконец, своей дочери Саюри, двухлетней «маленькой лилии в океане» (как переводится ее имя с японского и сингальского языков), за то, что она сохраняла рассудок своего папы, напоминая ему каждый день о самых важных вещах в жизни.

I

Постоянные изменения

Мы должны стать теми переменами,
которые мы хотим видеть в мире.

Махатма Ганди

Те, кто хочет пребывать в вечном блаженстве
мудрости, должны часто меняться.

Конфуций

Без изменений что-то спит внутри нас
и редко просыпается. Спящий должен проснуться.

Фрэнк Герберт

Жизнь принадлежит живущим.
И тот, кто живет, должен быть готов к переменам.

Иоганн Вольфганг фон Гете

Если бы потребовалось описать шаблоны проектирования одним предложением, мы бы сказали, что это *инструменты, помогающие справиться с постоянными изменениями* при проектировании и разработке программного обеспечения. Если вы посмотрите на разные шаблоны проектирования, представленные в книге, вы увидите, что все они оптимизированы для того, чтобы позволить программисту изменять и многократно использовать большую часть созданного им кода. Ключевыми понятиями являются изменение и гибкость. Заданная тема будет еще много раз рассматриваться в нашей книге. А часть I предоставляет общую информацию, необходимую для понимания и использования книги.

Для работы с шаблонами проектирования вам необходимо знать основные принципы объектно-ориентированного программирования (ООП). Если вы с ними не знакомы, хорошенько изучите главу 1. Ее заключительная часть переходит к обсуждению некоторых базовых для шаблонов проектирования вещей, их понимание поможет вам лучше разобраться в следующих главах, посвященных конкретным шаблонам.

Даже если вы уже знаете такие основополагающие понятия ООП, как абстрактность, инкапсуляция, наследование и полиморфизм, это еще не означает, что вы хорошо применяете ООП на практике. Чтобы считаться профессиональным проектировщиком и разработчиком программных приложений, вам необходимо уметь проектировать их так, чтобы их можно было легко обслуживать и развивать согласно новым требованиям. Другими словами, вам нужно создавать приложения, отражающие реальный мир. Инструменты, применяемые вами сегодня для разработки веб-приложений, должны позволять вам легко проводить изменения и обновления ваших приложений, а также повторно использовать имеющиеся наработки. В противном случае ваши приложения будут неспособны к адаптации под требования текущего дня.

Шаблоны проектирования дают образцы объектно-ориентированного проектирования приложений, которые позволяют справиться с изменениями в программах посредством различных средств ООП. И если вы думаете об использовании ООП как о методе создания проектов, способном помочь вам разобраться с переделками в приложениях, повторно использовать имеющиеся наработки и добавить гибкости вашим программам, то вы уже начинаете мыслить в духе шаблонов проектирования.

Глава 1 «Объектно-ориентированное программирование, шаблоны проектирования и язык ActionScript 3.0».

1

ООП, шаблоны проектирования и язык ActionScript 3.0

Приучите себя исследовать замысел действий людей и размышляйте об их возможных последствиях как можно чаще; а чтобы такой анализ стал еще более значительным, применяйте его в первую очередь по отношению к самим себе.

Марк Аврелий

История жизни личности – это прежде всего подстраивание под шаблоны и стандарты, традиционно принятые в обществе.

Рут Бенедикт

На нижнем когнитивном уровне находятся процессы восприятия, или, вообще говоря, интуитивные процессы, воспринимающие объекты в оригинале.

Эдмунд Гуссерль

Удовольствие от хорошей работы

Основной идеей применения шаблонов проектирования является выделение некоторого набора таких шаблонов и решение с его помощью всех похожих задач. Кроме того (можно сказать, неразрывно с этим), шаблоны являются некоторым стандартом хороших, общепринятых практик объектно-ориентированного программирования (ООП). Поэтому мы не можем (да и не хотим) отделять шаблоны проектирования от ООП. Отвечая на вопрос, зачем нужно использовать стандартные шаблоны проектирования, мы в действительности размышляем над вопросом, зачем нужно использовать ООП. Традиционный ответ на оба этих вопроса состоит в том, что благодаря шаблонам и ООП разработчики внутри одной

команды получают возможность разговаривать между собой на одном языке. Более того, используя объектно-ориентированные идеологии и практики программирования, проще справляться со сложностями, связанными с программистскими задачами, требующими разделения труда.

Кроме как для решения вопросов координации работ в больших проектах, программисты используют ООП и шаблоны проектирования для последующего внесения изменений в свои программные системы. Одной из их важнейших, ключевых характеристик шаблонов является возможность с их помощью гораздо легче изменять реализованные программы. Чем длиннее ваша программа и чем больше времени вы потратили на ее разработку, тем сложнее будет вносить в нее изменения и тем более непредсказуемыми будут последствия таких действий. Изменение кода в программе может иметь для нее разрушительные последствия, подобные распусканию свитера в результате вытягивания из него одной нити. Шаблоны проектирования и следование правилам ООП значительно облегчают внесение изменений в сложные программы, а также уменьшают объем этих изменений и снижают вероятность возникновения глобальных проблем.

Упрощение решения задач координации работ в команде разработчиков и изменения и поддержки существующего приложения уже является достаточным основанием для изучения шаблонов проектирования. Однако представим, что ваша программа относительно короткая, вы работаете самостоятельно и вас особо не волнуют дальнейшие изменения в вашей программе. В таком случае бывает быстрее просто переписать программу заново. Зачем вам тогда вообще нужно знать про шаблоны проектирования?

Помимо того обстоятельства, что ActionScript 3.0 базируется на ECMAScript и, вероятно, не будет так сильно меняться с каждым новым релизом Flash или Flex, как в прошлом, у вас может быть личная причина для изучения шаблонов проектирования. Александр Нахимовский (Alexander Nakhimovsky) и Том Майерс (Tom Myers) писали в своей книге, посвященной ООП и JavaScript (издательство Wrox, 1998), о ценности удовольствия от качественно проделанной работы. Как и при любом другом занятии, будь то катание на скейтборде или строительство дома, его хорошее выполнение приносит человеку радость. Под словами «сделать хорошо» мы вовсе не подразумеваем некий навязчивый перфекционизм, главным образом потому что он часто ведет к полному параличу во всех делах. Мы скорее говорим о такой работе, в ходе и по итогам которой вы испытываете удовлетворение искусного мастера от творческого процесса и от верных результатов.

Последовательное и процедурное программирование

Если вы никогда не слышали о *последовательном программировании*, то, скорее всего, именно этим видом программирования вы и занимае-

тесь. Большинство непрофессиональных программистов просто пишут в своей программе одно выражение за другим, и при правильной последовательности таких выражений она работает вполне сносно. Однако по мере того как программы все более и более усложнялись, у разработчиков образовывались невообразимые кучи кода, называемые часто *спагетти-кодом* (из-за такой же запутанности). Во избежание подобного эффекта программисты стали организовывать свои программы в виде множеств отдельных процедур и правил их использования. Это и стало началом *процедурного программирования*. Использование подпрограмм вместо неизбежных выражений GOTO, переносящих нас в различные места программы, позволило разбить поток исполнения команд на отдельные модули с соответствующими вызовами GOSUB/RETURN с сохранением опрятного вида всего кода..

Команда RETURN имела раньше немного другое значение, чем сейчас. Ранее RETURN означала возврат на то место в последовательности кода программы, откуда сработала команда GOSUB. В языке ActionScript команда return означает, что данная операция (метод или процедура) высылает назад некоторую свою информацию.

Из процедурного программирования к нам пришло понятие *области видимости*, в которой переменные функций или подпрограмм могут быть использованы программистом повторно, причем ни одна из процедур не повлияет на другую.

Подавляющее большинство языков программирования в наши дни считается процедурными в том смысле, что они концептуально и синтаксически поддерживают данный подход. Различные версии языка Basic являются процедурными, как и языки типа ColdFusion, PHP и Perl. Однако процедурные языки типа C++, ECMAScript (ActionScript 3.0) и Ada считаются скорее объектно-ориентированными языками программирования. Языки типа Java относят к чисто объектно-ориентированным. Не углубляясь сильно в детали, скажем, что язык Java – чисто ООП-язык, поскольку единственным видом процедур в Java является метод класса. Его конструкция вынуждает процедуры работать как методы класса и не позволяет другим процедурам действовать вне структуры классов.

Возможно, вы удивитесь, узнав, насколько острыми бывают дискуссии на тему, является некоторый язык программирования объектно-ориентированным или нет. В отношении этого вопроса существует два варианта критериев. Один из них весьма либерален и позволяет любому языку, в котором есть некие возможности генерировать ООП-код, считаться ООП-языком. (ActionScript 3.0 относится к такого рода языкам.) Согласно другому, гораздо более жесткому варианту *только те* языки, в которых все процедуры являются исключительно методами классов, могут быть допущены в эксклюзивный клуб ООП-языков. Оба этих варианта имеют свои весомые аргументы. Мы обойдем эту дис-

куссию и не будем принимать в ней ничью точку зрения, однако мы отметим, что сторонники обеих вполне согласны с тем, что с помощью процедурных языков программирования можно создавать хороший объектно-ориентированный код.

Короче говоря, вышесказанное не означает, что на других языках программирования невозможно писать настоящие ООП-программы. Разработчики с успехом их реализовывали еще задолго до появления Java. Некоторые языки программирования, особенно использующие структуры классов и перегружаемые методы типа ActionScript 3.0, являются более близкими к ООП-концепциям, чем другие. Язык ActionScript неуклонно приближался к ним по мере своего развития от набора нескольких синтаксических выражений до стандарта ECMAScript.

Переход к ООП

Переход от последовательного или процедурного к объектно-ориентированному программированию – это нечто большее, чем просто выбор языка, предоставляющего вам такую возможность (такого, например, как Java). Определенные изменения в языке программирования могут сделать его более пригодным для применения ООП, даже если он не является чисто ООП-языком согласно некоторым критериям. В следующих разделах мы кратко опишем некоторые новые особенности во Flash CS3, изменяющие способ использования ActionScript.

Код на кнопках и мувиклипах

Как и ветераны ActionScript, начинавшие еще с написания небольших скриптов, использующих команду `on` и ассоциированных с объектами кнопок и клипами (`MovieClip`, `Button`), вы, вероятно, знаете, что последняя версия Flash вообще не содержит механизмов для написания подобного встроенного кода.

Встроенные автоматы: хотя большинство программистов приветствовало кончину внутренних сценариев для кнопок и клипов, один более проницательный из них заметил, что ранее во Flash были встроенные автоматы состояний. Джонатан Кей (Jonathan Kaye), соавтор книги «Flash MX for Interactive Simulation: How to Construct and Use Device Simulations» (издательство Delmar Learning, 2002), обратил внимание на то, что такие скрипты кнопок и клипов часто служили для создания внутренних автоматов в программах. Каждая кнопка или клип могли служить изолированным, чувствительным к контексту триггером для изменения их состояния. (О том, как шаблоны проектирования реализуют объекты с изменяющимся состоянием, читайте в главе 10.)

Вообще утрата внутренних сценариев для кнопок и клипов воспринимается как благоприятная возможность для улучшения программ, осо-

бенно для применения ООП. Отслеживание в программе всех изолированных скриптов для кнопок и клипов могло причинять головную боль даже в случае относительно небольшого приложения. А при построении крупных программ, в которых исключительно полезны ООП и шаблоны проектирования, множество разнообразного кода на кнопках и клипах превращало эту проблему из головной боли в сущий кошмар. Поэтому, изучая шаблоны проектирования, радуйтесь, что вам не нужно думать о маленьких сценариях, изолированных в клипах и кнопках.

Код на временной диаграмме

Другой тип сценариев, который вы будете видеть здесь гораздо реже, чем во Flash, – сценарии, привязанные к временной диаграмме (Timeline). Размещение сценариев на временной диаграмме, возможно, стоило убрать в первую очередь, однако оно оказалось весьма удобным. В ActionScript 2.0 вы могли поместить сценарий в класс и вызвать его из сценария на временной диаграмме; поэтому временные сценарии в действительности использовались только для вызова сценариев классов, размещенных в файлах ActionScript (*.as*). Для таких случаев в свойствах *.fla*-файлов Flash CS3 имеется поле, где вы можете добавить имя вызываемого класса. (Смотрите следующий раздел.) Таким образом, если вы хотите лишь вызвать программу и скомпилировать ее в *swf*-файл, то вам вовсе не требуется использовать в своем коде временную диаграмму.

Flash CS3 не отказался от кода на временной диаграмме. Вы все еще можете им пользоваться, но в данной книге мы будем применять его ограниченно, только для клипов, вызываемых из класса, находящегося вне классов клипов или кнопок. (Смотрите раздел «Классы кнопок и клипов».)

Класс документа

Вам не придется (если вообще придется) писать много кода для временной диаграммы при использовании версии ActionScript 3.0. Теперь не обязательно применять объекты из временной диаграммы; вы можете компилировать ваши *.as*-файлы, просто вводя имя класса, с помощью которого вы хотите запустить ваше приложение. Рисунок 1.1 показывает, как использовать поле класс документа (Document class) из меню Свойства документа (Properties), чтобы ввести имя класса для запуска.

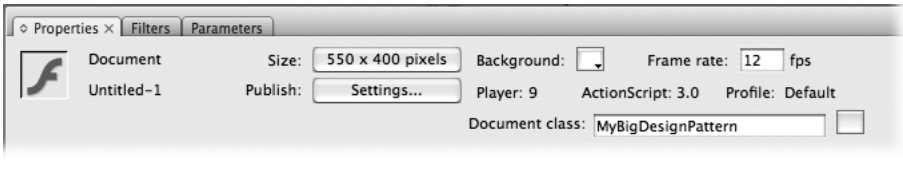


Рис. 1.1. *Поле класса документа*

Вы по-прежнему можете использовать временную диаграмму, однако если у вас нет на это каких-то особых причин, в этом нет необходимости. Большинство примеров в данной книге используют объекты класса `Sprite` вместо класса `MovieClip`. У объекта класса `Sprite` нет своей временной диаграммы в отличие от объекта класса `MovieClip`, поэтому использование `Sprite` спасает от связанных с ней дополнительных забот.

Классы кнопок и клипов

Во Flash CS3 объекты `MovieClip` и `Button`, которые вы создаете с помощью окна диалога Символ (Symbol) и сохраняете на панели Library (Библиотека), могут быть использованы и в ActionScript 3.0. В отличие от ActionScript 2.0, где символы клипов и кнопок могли быть только ассоциированы с другим классом, во Flash CS3 *они сами могут быть классами*. Имя объекта, введенное в окне Name (Имя) при создании этого объекта, становится его именем класса. (В последних версиях ссылки на клипы делаются через *имя экземпляра (instance name)*; то есть вы все еще можете их создавать, но уже в другом контексте.)

Преимуществом такого нового подхода является то, что теперь любые объекты-символы могут быть представлены, как и любой другой класс, в коде, если эти символы есть в Библиотеке. Не обязательно сразу размещать их на сцене. Они могут быть динамически созданы и выведены на дисплей, как любой другой динамический объект. Более того, объекты, содержащие клипы или кнопки, могут рассматриваться как свойства наравне со всеми прочими классами.

Хотя данная книга вовсе не является введением во Flash CS3, рассмотрение одного примера нового способа создания класса с клипами и кнопками может быть полезно как для новичков, так и для опытных пользователей Flash. Итак, вы можете опробовать новую возможность, выполнив следующие шаги:

1. Откройте новый документ Flash и сохраните его как *rocket.fla*.
2. Выберите из меню Insert → New Symbol (Вставить → Новый символ) и откройте диалоговое окно создания нового символа. В окне Name (Имя) введите `Rocket` и нажмите на `OK`, чтобы перейти в режим редактирования символа.
3. В режиме редактирования символа нарисуйте на сцене ракету с размерами W=89 (ширина), H=14 (высота), как показано на рис. 1.2. По

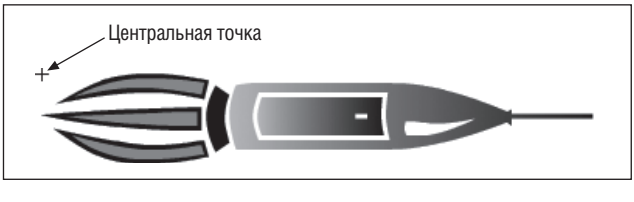


Рис. 1.2. *Рисунок ракеты*

завершении рисования поместите ее изображение в позицию X=0, Y=0. Щелкните на иконке Сцены 1, чтобы выйти из режима редактирования символа.

4. Выберите из меню Insert → New Symbol (Вставить → Новый символ), чтобы открыть диалоговое окно конвертации в символ (Convert to Symbol). Введите FireRocket в поле Name (Имя), выберете клип (Movie clip) в качестве типа и поставьте галочку напротив Export for ActionScript (Экспорт для ActionScript). После этого ваше диалоговое окно станет подобным изображению на рис. 1.3. Заметьте, базовым классом является `flash.display.MovieClip`. Базовый класс представляет собой ссылку на пакет, необходимый ActionScript для отображения объекта `MovieClip`. Нажмите OK, чтобы перейти в режим редактирования символа.

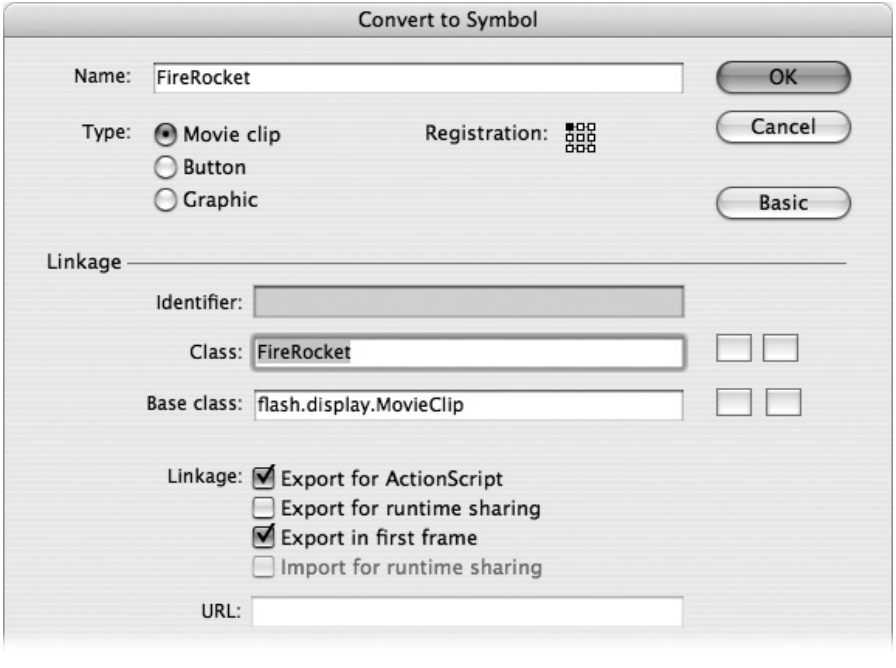


Рис. 1.3. *Установки для класса MovieClip*

5. Перетащите копию клипа ракеты из Библиотеки на центр сцены. Сдвиньте центральную точку клипа к хвосту ракеты и установите ее на позиции X=0, Y=0.
6. Кликните на 40-й кадр временной диаграммы и нажмите F5, чтобы создать на ней 40 кадров. Опять кликните на 40-м кадре и нажмите F6, чтобы создать ключевой кадр. Кликните на ключевом кадре 40-го кадра и перетащите ракету на позицию X=400, Y=0.
7. Кликните на первом ключевом кадре и в выпадающем окне инспектора свойств выберите Motion (Движение). Вы должны увидеть синюю

стрелку на временной диаграмме. Подвигайте указатель кадра слева направо и убедитесь, что движение ракеты реализовано должным образом. Рисунок 1.4 показывает то, что вы должны увидеть.

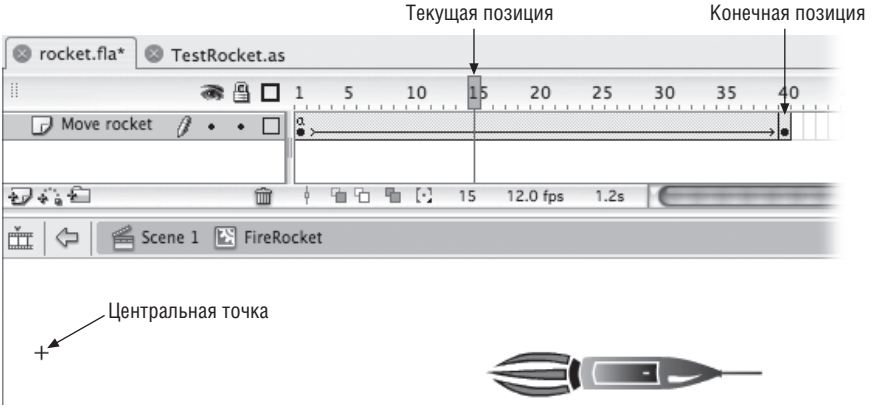


Рис. 1.4. *Ракета в движении*

8. Откройте панель Actions (Действия). Кликните на пустой области сцены, чтобы у вас точно не было выбранного объекта, и затем кликните на первом кадре. В панели действий введите выражение *stop()*. Сохраните файл *Rocket.fla*.
9. Откройте новый файл ActionScript и сохраните его под именем *TestRocket.as* в том же каталоге, где содержится файл *Rocket.fla*. Введите скрипт из примера 1.1 в файл *TestRocket.as* и сохраните файл еще раз.

Пример 1.1. TestRocket.as

```
package
{
    import flash.display.Sprite;
    public class TestRocket extends Sprite
    {
        private var fireRocket:FireRocket;
        public function TestRocket( )
        {
            fireRocket=new FireRocket( );
            fireRocket.x=50;
            fireRocket.y=100;
            addChild(fireRocket);
            fireRocket.gotoAndPlay(2);
        }
    }
}
```

10. И, наконец, откройте файл *Rocket.fla*, в поле класса документа на панели свойств наберите `TestRocket` и сохраните файл. Затем протестируйте ваш минифильм, нажав Ctrl+Enter (Command+Return на Макинтошах). Вы должны увидеть ракету, летящую слева направо по экрану и потом возвращающуюся на исходную позицию.

Более или менее традиционное использование Flash для создания клипов все еще остается важной областью применения ActionScript, но кое-что изменилось. Вы больше не можете присоединять к клипу класс, как это делалось в предыдущих версиях. Однако при создании приложений, использующих шаблоны проектирования, вы по-прежнему можете интегрировать в них различные объекты, созданные в среде проектирования (IDE) Flash. Так что, хотя ActionScript и сделал прыжок к ECMAScript, он не оторвался от своих корней в анимационной графике.

Основы ООП

Если вы хорошо знакомы с ООП и успешно применяете его на практике, вам, вероятно, можно пропустить этот раздел или только бегло просмотреть его на случай, не добавили ли мы что-то новое об ООП или нет ли в нем нового материала, касающегося языка ActionScript. Далее в этой главе мы подробно рассмотрим хорошо зарекомендовавшие себя концепции ООП, на которых базируются все шаблоны проектирования. Эти концепции рассчитаны на хорошее понимание основ ООП. К сожалению, наше краткое их обсуждение не сможет обеспечить вам достаточной глубины их понимания. Если это ваше первое знакомство с ООП, то вам, несомненно, потребуется пополнить ваши знания по данной теме с помощью хороших учебников, посвященных ООП.

На протяжении всей книги вы будете сталкиваться с примерами того, как шаблоны проектирования используют различные базовые принципы ООП. В каждой ее главе содержится раздел, посвященный некоторым ключевым концепциям ООП в шаблонах; в данной вводной главе вы встретитесь лишь с первым из множества случаев их описания. Это сделано нами специально. Мы полагаем, что рассмотрение концепций ООП с различных точек зрения поможет вам лучше разобраться во многих их нюансах. Для нас самих было удивительно, что разные шаблоны проектирования настолько по-разному освещают одни и те же концепции ООП и помогают глубже понять их смысл.

Для начала мы вспомним четыре базовые концепции ООП:

- Абстрактность
- Инкапсуляция
- Наследование
- Полиморфизм

Каждая из них нуждается в осмыслении; если вы новичок в ООП, не мучайтесь, стараясь понять их с первого раза. Мы будем еще много раз возвращаться к этим концепциям в главах, посвященных конкретным шаблонам проектирования.

Абстрактность

В общем абстракция – это модель или образец. Вы не знаете все ее детали, а только некоторые основные параметры, которые могут быть уточнены и дополнены. При этом абстракция достаточно точна, чтобы отличаться от других. Рассмотрим для примера две свободные вакансии в вашей компании: веб-дизайнера и программиста. В объявлении о них вы скорее укажете основные характеристики кандидатов, требуемые для выполнения соответствующей работы, чем какие-то индивидуальные их особенности. Тогда у вас получатся две абстракции, представляющие две различные позиции:

Открыты две вакансии:

- Программист
  - Имеющий опыт работы в команде
  - Имеющий опыт программирования межплатформенного программного обеспечения (ПО) и баз данных
  - Знакомый с языком ECMAScript
  - Имеющий навыки объектно-ориентированного программирования и знание шаблонов проектирования
- Веб-дизайнер
  - Имеющий опыт создания графических сайтов
  - Знакомый с анимацией
  - Имеющий опыт работы с векторной графикой
  - Практикующий клиенто-ориентированный подход

Здесь легко заметить разницу между двумя позициями и основными предъявляемыми к ним требованиями (свойствами); но их детали остаются открытыми для заполнения. Веб-дизайнер вряд ли будет претендовать на работу программиста, равно как и программист – на позицию веб-дизайнера. Однако кандидаты на эти должности будут обладать множеством различных дополнительных свойств и навыков, которые и станут конкретными деталями для позиций. Например, один из программистов может иметь опыт работы с PHP и/или с базой данных MySQL, в то время как другой программист может знать ASP.NET, C# и MS SQL. Таким образом, абстракция дается в описании должности, а деталями служат навыки и опыт работы каждого из кандидатов.

В своей книге «Object-Oriented Design with Applications» (издательство Benjamin/Cummings) Гради Буч (Grady Booch), один из пионеров в об-

ласти шаблонов проектирования, дал следующее определение абстракции, одновременно краткое и понятное:

> Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Под определение Буча очень хорошо подходят наши описания двух свободных позиций. Они содержат в себе важнейшие характеристики кандидатов на эти вакансии и позволяют четко отличить одну должность от другой.

Абстракции в ActionScript 3.0

Обращаясь теперь к абстракциям в программировании на ActionScript, в качестве примера мы рассмотрим простой видеопроигрыватель. Он будет состоять из некоторых необходимых нам элементов, поэтому мы начнем с их перечисления как абстракций:

- Сетевое соединение
- Видеоэкран
- Поток видео
- *flv*-файл для воспроизведения

Если нам удастся правильно собрать вместе все эти части, мы сможем проигрывать видео. Однако мы хотим начать нашу деятельность не с абстракций, а с чего-то конкретного, работающего уже сейчас. Введите код из примера 1.2 и сохраните его в файле с именем из заголовка примера:

Всюду в книге, за исключением нескольких случаев, имена из названий примеров используются как имена файлов для сохранения их кода.

Пример 1.2. PlayVideo.as

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.display.Sprite;

    public class PlayVideo extends Sprite
    {
        public function PlayVideo()
        {
            var nc:NetConnection=new NetConnection( );
            nc.connect(null);
            var ns:NetStream = new NetStream(nc);
```

```
            var vid:Video=new Video( );
            vid.attachNetStream(ns);
            ns.play("adp.flv");
            addChild(vid);
            vid.x=100;
            vid.y=50;
        }
    }
}
```

Вам потребуется *flv*-файл с именем *adp.flv*; любой *flv*-файл с таким именем будет работать. Откройте новый Flash файл, введите `PlayVideo` в поле класс документа и протестируйте его.

Чтобы сделать наш код абстрактным, нужно убрать из него все конкретные значения, за исключением `null` из метода `NetConnection.connect()`. (Его можно было бы передавать как строку, но для простоты мы его оставляем.) В примере 1.3 приводится то же приложение, только абстрагированное до «описания» того, что ему требуется для работы.

Пример 1.3. PlayVideoAbstract.as

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.display.Sprite;

    public class PlayVideoAbstract extends Sprite
    {
    public function PlayVideoAbstract(nc:NetConnection,
        ns:NetStream,vid:Video,flick:String,xpos:uint,ypos:uint)
        {
            nc=new NetConnection();
            nc.connect(null);
            ns= new NetStream(nc);
            vid=new Video();
            vid.attachNetStream(ns);
            ns.play(flick);
            vid.x=xpos;
            vid.y=ypos;
            addChild(vid);
        }
    }
}
```

Здесь все конкретные значения (за исключением `null`) заменены их абстракциями для *описания* работы данного объекта. Но, как и требования в описании вакансии, эти абстракции в классе `PlayVideoAbstract` необходимо заполнить чем-то конкретным. Все частности помещены в один длинный набор параметров:

```
    PlayVideoAbstract(nc:NetConnection,ns:NetStream,
        vid:Video,flick:String,xpos:uint,ypos:uint)
```

Эти абстрактные параметры в конструкторе класса позволяют нам добавлять в него любой конкретный, нужный нам элемент, включая имя файла видео для проигрывания. Пример 1.4 показывает, как конкретные элементы используются в абстрактном классе.

Пример 1.4. PlayAbstract.as.

```
package
{
    import flash.display.Sprite
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;

    public class PlayAbstract extends Sprite
    {
        private var conn:NetConnection;
        private var stream:NetStream;
        private var vid:Video;
        private var flick:String="adp.flv";

        public function PlayAbstract( )
        {
            var playIt:PlayVideoAbstract=new PlayVideoAbstract(conn,stream,vid,
flick,100,50);
            addChild(playIt);
        }
    }
}
```

Все, что делает данный класс, – это создает один объект класса `PlayVideoAbstract` и размещает его на сцене. Закрытые переменные служат для передачи большинства конкретных значений необходимых параметров. Константы указывают горизонтальную (`x`) и вертикальную (`y`) позиции для проигрывания видео. Чтобы протестировать данный пример, просто измените во Flash для *fla*-файла имя в поле класса документа на `PlayAbstract`.

Почему абстракции так важны

Можно выделить две основные причины, по которым абстракции так важны и для ООП, и для шаблонов проектирования. Во-первых, абстракции позволяют нам сосредоточиться на тех независящих от своих деталей частях, которые требуются для решения наших задач, а не заниматься мелочами. Означает ли это, что детали игнорируются? Вовсе нет. Детали добавляются, когда они действительно нужны. Например, в примере в предыдущем разделе конкретный видеофайл нам не важен. Важно лишь, что какое-то имя видеофайла (конкретная деталь) будет

нам передано, когда мы будем готовы проиграть это видео. Так же как нам не нужно строить целый кинотеатр ради просмотра одного фильма, нам не нужно создавать класс ради просмотра одного видеофайла.

Во-вторых, абстракции дают нам большую гибкость в наших решениях. Если вам показалось, что реализация нашего приложения из примера 1.2 предыдущего раздела была проще и потребовала меньше кода и классов, вы абсолютно правы. Представьте теперь, что вам требуется разместить на сцене сразу четыре видео. Тогда вместо того, чтобы переписывать свой класс еще три раза, вам нужно будет лишь создать четыре объекта этого абстрактного класса. Другими словами, вторая реализация, использующая абстракции, гораздо более гибкая к изменениям. Мы можем не только создавать несколько видеообъектов, но и легко изменять в них имена видеофайлов для воспроизведения.

Инкапсуляция

Инкапсуляция – то, что делает код объекта целостным объектом. Если у вас имеются в наличии хвост, четыре ноги, холодный нос и звук лая, это еще не означает, что у вас есть собака. У вас есть только набор исходных частей, из которых она состоит. Если вы соберете все их вместе и получите собаку, то хотя вы и будете помнить, что все ее части являются частями, вы станете воспринимать собаку как уникальное целое, не думая больше о частях. То есть объектом будет уже собака, а не ее части, сложенные вместе. Инкапсуляция производит схожий эффект на наборы операций и свойств.

Понятие инкапсуляции часто упоминается вместе с такими понятиями, как *компонент* и *модуль*. В контексте ООП инкапсуляцию очень часто называют *черным ящиком*, в том смысле, что вы можете наблюдать внешние проявления объекта, но не можете видеть, как все устроено и работает у него внутри. На самом деле многие вещи, с которыми мы встречаемся в нашей жизни, являются для нас черными ящиками, например, собака. Мы можем смотреть, как собака делает множество разнообразных вещей, мы можем контактировать с ней, но мы толком не знаем (да нам это обычно и не важно), как при этом работает ее организм, поскольку она не прозрачная. Собака – черный ящик.

Хорошей стороной концепции черного ящика является то, что нам не приходится заботиться о его внутренней работе или устройстве. Нам необходимо лишь уметь правильно с ним обращаться, будучи подкрепленными знанием, что пока он действует так, как нам нужно, внутри у него все в порядке.

Сокрытие внутренних данных «во избежание»

Чтобы понять, почему нам важно инкапсулировать свои данные, мы рассмотрим две программы. Одна будет без инкапсуляции, что приве-

дет к нежелательным последствиям, а вторая – с инкапсуляцией, что поможет нам избежать странных результатов ее работы.

Если вы создаете объект Собака, то вы, наверное, захотите включить в него операции, дающие собаке возможность как-то общаться с внешним миром. Для иллюстрации мы создадим метод под названием `dogTalk` («собачий язык»), который позволит собаке издавать некоторые звуки. Собака будет у нас общаться следующими способами:

- Гавкать
- Скулить
- Выть
- Рычать

Мы начнем с *плохого* применения ООП, чтобы показать вам, как в словаре собаки могут образовываться нежелательные выражения. Код примера 1.5 является неинкапсулированным и в будущем способен поставить вашу собаку в неудобное положение.

Пример 1.5. NoEncap.as

```
package
{
    // Это плохое ООП – без инкапсуляции
    import flash.text.TextField;
    import flash.display.Sprite;

    public class NoEncap extends Sprite
    {
        public var dogTalk:String="Woof, woof!";
        public var textFld:TextField=new TextField( );

        public function NoEncap( )
        {
            addChild(textFld);
            textFld.x=100;
            textFld.y=100;
        }
        function showDogTalk( )
        {
            textFld.text=dogTalk;
        }
    }
}
```

В классе, имеющем свойства черного ящика, у вас *не должно быть* возможности менять его внутреннюю работу. Но этот класс, как вы увидите, открыт для таких изменений. С инкапсулированным объектом вы можете взаимодействовать только через его интерфейс, и вы не долж-

ны позволять приложению вносить внутрь него какие-либо изменения. Код в примере 1.6 вламывается внутрь нашего объекта и меняет его нежелательным образом.

Пример 1.6. TestNoEncap.as

```
package
{
    import flash.display.Sprite;

    public class TestNoEncap extends Sprite
    {
        public var noEncap:NoEncap;
        public function TestNoEncap()
        {
            noEncap=new NoEncap();
            noEncap.dogTalk="Meow";
            noEncap.showDogTalk();
            addChild(noEncap);
        }
    }
}
```

Откройте новый Flash-файл и в поле класс документа введите `TestNoEncap`. Когда вы запустите этот файл, вы увидите на экране «Meow» («Мяу»). Такой ответ от вашего объекта-собаки никуда не годится. Собаки не мяукают, а кошки не лают. Тем не менее подобное вполне может произойти, если вы не инкапсулировали свой класс. А если вы умножите эту проблему на количество всех неинкапсулированных классов в вашем приложении, то вы можете представить себе, какой ужасный беспорядок в результате у вас получится. Так что давайте-ка исправлять ситуацию.

Приватные переменные

Простейшим способом обеспечить инкапсуляцию ваших данных является использование приватных (или «закрытых») переменных. Ключевое слово `private` языка ActionScript 3.0, будучи примененным к переменным, константам или методам (функциям), гарантирует, что только класс, в котором они были определены или объявлены, будет иметь к ним доступ. Оно не только закрывает от них наши приложения, пытающиеся присвоить им какие-либо значения, но и исключает подклассы. (В этом состоит отличие от ActionScript 2.0; поэтому будьте внимательны при преобразовании приложения с ActionScript 2.0 на ActionScript 3.0.)

Давайте посмотрим, как слово `private` изменит работу нашего приложения. В примере 1.7 к переменным класса `NoEncap` были добавлены атрибуты `private`.

Пример 1.7. Encap.as

```
package
{
    // Это хорошее ООП – инкапсуляция присутствует
    import flash.text.TextField;
    import flash.display.Sprite;

    public class Encap extends Sprite
    {
        private var dogTalk:String="Woof, woof!";
        private var textFld:TextField=new TextField();

        public function Encap()
        {
            addChild(textFld);
            textFld.x=100;
            textFld.y=100;
        }
        function showDogTalk()
        {
            textFld.text=dogTalk;
        }
    }
}
```

Небольшие изменения нужно внести и в тестовый файл. Его реализация суперкласса должна быть изменена. В примере 1.8 показан новый тестовый класс, `TestEncap`, для класса `Encap`.

Пример 1.8. TestEncap.as

```
package
{
    import flash.display.Sprite;

    public class TestEncap extends Sprite
    {
        public var encap:Encap
        public function TestEncap()
        {
            encap=new Encap();
            encap.dogTalk="Meow";
            encap.showDogTalk();
            addChild(encap);
        }
    }
}
```

Попробуйте его запустить, изменив имя класса документа на `TestEncap`. Вы получите сообщение об ошибке от компилятора:

```
Line 11: 1178: Attempted access of inaccessible property dogTalk through
a reference with static type Encap.
Source: encap.dogTalk="Meow";[1]
```

Оно показывает, что источником ошибки является строка

```
encap.dogTalk="Meow";
```

Такая ошибка отражает тот факт, что произошла попытка доступа к приватной переменной извне ее класса. Чтобы исправить код, закомментируйте неправильную строку:

```
//encap.dogTalk="Meow";
```

Теперь опять все протестируйте. На этот раз все должно сработать нормально, и вы не получите объект-собаку, говорящую «Meow» («Мяу»). Вы увидите «Woof, woof» («Гав, гав»).

Вам может показаться, что приватные переменные сильно ограничивают вас в ваших возможностях. А вдруг вы захотите, чтобы ваш объект-собака дополнительно завывал, рычал или поскуливал? Как вносить такие изменения динамически, в ходе выполнения программы? Защита инкапсулированного объекта от неправильных действий с ним это одно дело, но как теперь заставить такой объект принимать приемлемые значения для переменных?

Многозначность термина «интерфейс»

В данной книге вы будете встречать слово *интерфейс* в различных контекстах, и каждый из них будет придавать ему несколько иное значение. (Вот спасибо!) Вы, наверно, уже знакомы с такими терминами, как UI (интерфейс пользователя) или GUI (графический интерфейс пользователя), которые означают различные средства для взаимодействия пользователя с программой. Например, кнопка – очень распространенный элемент интерфейса пользователя во Flash. Когда вы нажимаете на кнопку, происходит некоторое запланированное действие. Вы можете не иметь представления (или не интересоваться), как именно это делается, но вы знаете, что если нажать на кнопку, то будет проиграно видео, или появится следующая страница, или начнется некоторая анимация. Итак, если вы понимаете основную концепцию пользовательского интерфейса, то вы вполне в состоянии понять и то, как работает интерфейс с объектом.

Когда вы работаете с пользовательским интерфейсом, само приложение является для вас черным ящиком; неважно, совершаете вы с его помощью покупки на eBay или используете его как текстовый редактор. Если вы следуете определенным правилам и используете элементы

[1] Строка 11, ошибка 1178: Доступ к вероятно неопределенному свойству `dogTalk` посредством ссылки со статическим типом `Encap`. Источник: `encap.dogTalk="Meow";` – *Прим. науч. ред.*

интерфейса пользователя надлежащим образом, вы получите от приложения то, что хотели. Инкапсулированный объект также является для вас черным ящиком, а его интерфейс определяет, какими способами вы можете взаимодействовать с ним из программы, то есть является аналогом пользовательского интерфейса для объекта.

Интерфейсы объекта и их сигнатуры хорошо описаны в книге «Design Patterns: Elements of Reusable Object-Oriented Software» (с. 13). *Сигнатурой* объекта называют имя его операции, ее параметры и тип результата. Рисунок 1.5 графически показывает типичный вид сигнатуры объекта.

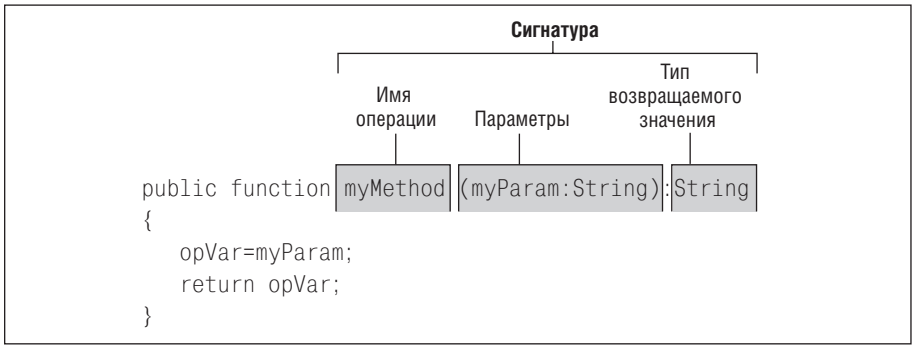


***Рис. 1.5.** Сигнатура объекта*

Все сигнатуры объекта, задаваемые его операциями, образуют его интерфейс. Тогда в данном контексте *интерфейс объекта состоит из способов доступа к его данным, списка его сервисов и способов их вызова*.

Постойте! Это еще не все! Далее в этой главе вы увидите ключевое слово `interface` из лексикона ActionScript 3.0, которое определяет совсем иное понятие *интерфейса*. В других разделах книги вы также обнаружите термин *supertype*, употребляемый в качестве синонима интерфейса. Все эти различные значения термина *интерфейс* будут объяснены нами в свое время.

Прежде чем совсем увязнуть в этих контекстно-зависимых определениях, хорошо бы нам посмотреть, как выглядит интерфейс инкапсулированного объекта. Следующий раздел посвящен как раз этому.

Получатели и установщики

Наиболее традиционным способом реализации инкапсуляции с возможностью некоторого внешнего доступа к объекту является использование интерфейса с получателями («getters») и установщиками («setters»). В таком случае объект контролирует и доступ к своим данным, и то, как их можно изменять. Вернемся к нашему примеру с объектом-собакой. Мы знаем, что у собак ограниченный словарь вы-

ражений и «Мяу» в него никак не входит. Следовательно, мы можем просто реализовать установщик для данного значения, который разрешит лишь выражения из ограниченного словаря собаки.

Метод-установщик предполагает некоторый параметр – переменную некоторого типа – и некоторый алгоритм, который что-то пропускает, а что-то запрещает. Однако многие установщики просто присваивают переданное им значение параметра переменной объекта независимо от ее значения. Алгоритм и все остальное в функции, что составляет метод, невидимо для приложения, и если вводится неправильное значение параметра или параметр неподходящего типа, то либо появляется сообщение об ошибке, либо значение не передается. Ниже приводится типичная структура установщика:

```
function setterMethod(parameter)
{
    if(parameter=="OK")
    {
        private variable = parameter;
    }
}
```

Таким образом, весь фокус тут в том, что приложение может передавать объекту любые данные для установщика, и, если они подходящие, то значение переданного параметра будет присвоено приватной переменной объекта. Это очень общее описание того, как действуют установщики, но оно отражает суть их работы.

По сравнению с установщиками методы-получатели устроены довольно просто. В примере 1.7 метод `showDogTalk()` является функцией-получателем. Задача метода-получателя – предоставить данные объекта внешней программе. Таким образом, хотя в примере и нет установщика, получатель там присутствует. Задача установщика заключается в том, что клиент получит только то, что должен получить.

В примере 1.9 приватной переменной `dogTalk` нельзя, как и ранее, присвоить произвольное значение. Однако для нее имеются методы установщика и получателя. И, как вы увидите, протестировав новый класс, `EncapSet`, приложение имеет доступ к этой переменной через интерфейс установщика.

Пример 1.9. EncapSet.as

```
package
{
    // Это ООП еще лучше –
    // помимо инкапсуляции используется интерфейс для объекта

    import flash.text.TextField;
    import flash.display.Sprite;
```

```
    public class EncapSet extends Sprite
    {
        private var dogTalk:String;
        private var textFld:TextField=new TextField();

        public function EncapSet()
        {
            addChild(textFld);
            textFld.x=100;
            textFld.y=100;
        }

        // Установщик (Setter)
        function setDogTalk(bowWow:String)
        {
            switch (bowWow)
            {
                case "Woof" :
                    dogTalk=bowWow;
                    break;

                case "Whine" :
                    dogTalk=bowWow;
                    break;

                case "Grrrr" :
                    dogTalk=bowWow;
                    break;

                case "Howl" :
                    dogTalk=bowWow;
                    break;

                default :
                    dogTalk="Not dog talk!";
            }
        }

        // Формируем результат
        function showDogTalk()
        {
            textFld.text=dogTalk;
        }
    }
}
```

Как видно из этого примера, метод-установщик допускает только четыре перечисленных в нем выражения для собаки. Все прочее (включая «Мяу») запрещено. Далее, в примере 1.10, показан тест для данного инкапсулированного объекта-собаки:

Пример 1.10. TestEncapSet.as

```
package
{
    import flash.display.Sprite;

    public class TestEncapSet extends Sprite
    {
        private var encapSet:EncapSet
        public function TestEncapSet()
        {
            encapSet=new EncapSet();
            encapSet.setDogTalk("Howl");
            encapSet.showDogTalk();
            addChild(encapSet);
        }
    }
}
```

Введите `TestEncapSet` в поле класс документа для *fla*-файла и запустите его. Как вы увидите, строка «Howl» («Выть») прекрасно проходит. А теперь протестируйте его еще раз, используя «Meow» («Мяу»). На этот раз вы увидите, что наш объект отвергает кошачий звук со словами «Not dog talk» («Собаки так не говорят»). Таким образом, мы убили двух зайцев одним выстрелом: реализовали инкапсуляцию и возможность оперировать с данными внутри инкапсулированного объекта.

Методы get и set

Еще одним способом управления инкапсуляцией и сохранения информации ваших объектов является применение специальных методов ActionScript 3.0 `get` и `set`. Некоторые программисты считают, что довольно неудобно создавать свои собственные получатели и установщики, как мы делали в предыдущем разделе, и предпочитают вместо этого пользоваться простотой работы *аксессора (accessor)* `get` и *мутатора (mutator)* `set`.

Аксессоры и мутаторы против Франкенштейна: термины *аксессор* и *мутатор* звучат как какие-то словечки из фильма ужасов, но на самом деле они применяются для описания получателей и установщиков. Аксессор (получатель) используется для доступа или *получения* информации, и здесь все понятно. Но что такое мутатор? Ну, если задуматься, то мутации означают некоторые изменения, а когда мы задаем новые значения чего-либо, мы, несомненно, их изменяем. Это тоже совершенно очевидно. Так что если вам посчастливится читать статью, посвященную шаблонам проектирования, пусть такие термины, как *аксессор* и *мутатор*, не сильно вас пугают.

Ключевым отличием использования методов `get` и `set` является отсутствие круглых скобок. Изменения (установка) выполняются не путем передачи новых значений. Получатели и установщики рассматриваются как некоторые свойства объекта, где значения приписываются или находятся через присваивание.

Примеры 1.11 и 1.12 покажут вам, как можно обезопасить ваш инкапсулированный объект с помощью методов получателя и установщика вида `get` и `set` соответственно.

Пример 1.11. FlowerShop.as (Цветочный магазин)

```
package
{
    public class FlowerShop
    {
        private var buds:String;

        public function FlowerShop():void {}

        // Получатель (Getter)
        public function get flowers():String
        {
            return buds;
        }

        // Установщик (Setter)
        public function set flowers(floral:String):void
        {
            buds=floral;
        }
    }
}
```

В примере 1.12 помните, что `flowers` – это метод, а не свойство объекта. Хотя установка и получение значений с его помощью выглядят точно так же, как установка и получение свойства.

Пример 1.12. SendFlowers.as (Послать цветы)

```
package
{
    import flash.display.Sprite;

    public class SendFlowers extends Sprite
    {
        public function SendFlowers()
        {
            var trueLove:FlowerShop = new FlowerShop();
```

```
            // Установить значение
            trueLove.flowers="A dozen roses";
            // Получить значение
            trace(trueLove.flowers);
            // Установить другое значение
            trueLove.flowers="And a dozen more....";
            // Получить измененное значение
            trace(trueLove.flowers);
        }
    }
}
```

Использование инкапсуляции в шаблонах проектирования

Этот раздел, посвященный инкапсуляции, получился достаточно длинным. Но причиной такого внимания к ней является важность инкапсуляции для хорошего ООП; она является неотъемлемым элементом шаблонов проектирования. Из 24-х первоначальных шаблонов проектирования только 4 относятся к классам, а остальные 20 – к объектам. При этом большинство шаблонов проектирования основываются, по сути, на композиции, а вовсе не на наследовании (которое обсуждается в следующем разделе).

Позднее в этой главе, в разделе «Преимущество композиции», вы увидите, как шаблоны проектирования создаются из нескольких разных объектов. Чтобы наши шаблоны при этом работали так, как нужно, инкапсуляция объектов в них просто необходима.

Наследование

Третьей ключевой концепцией хорошего ООП является *наследование*. Наследование означает способ, с помощью которого один класс наследует свойства, методы и события от другого класса. Если в классе А имеются методы X, Y и Z и класс Б является подклассом (расширяет) класса А, то в нем тоже будут эти методы. Наследование позволяет сэкономить много времени при разработке программных проектов и добавляет им функциональности. Если вы программировали в ActionScript 3.0, вы, вероятно, расширяли класс `Sprite`, как мы это делали в примерах с 1.1 по 1.10. Именно благодаря наследованию новые классы (т. е. подклассы), полученные из класса `Sprite`, приобретали всю его функциональность дополнительно ко всему, что добавляли в них вы.

Просмотр предков

Самое время разобраться, как работает наследование в ActionScript 3.0. Откройте вашу электронную документацию «ActionScript 3.0 Language Reference». В окне Packages (Пакеты) кликните на `flash.display`. В главном

окне, в котором появится информация о пакете flash.display, кликните на MovieClip в таблице классов. В самом верху страницы, посвященной классу MovieClip, вы увидите его путь наследования:

```
MovieClip → Sprite → DisplayObjectContainer → InteractiveObject →
DisplayObject → EventDispatcher → Object
```

Это означает, что класс MovieClip наследует все характеристики корневого объекта Object и всех остальных вплоть до объекта Sprite.

Пролистайте окно вниз до раздела Public Properties (Публичные свойства). Вы увидите там девять свойств. Кликните по ссылке Show Inherited Public Properties (Показать унаследованные публичные свойства). Теперь вы должны увидеть еще 43 дополнительных свойства! Итак, из 52 свойств класса MovieClip только 9 являются уникальными для него.[1] Все остальные им унаследованы. Те свойства и методы, которые мы сами добавляли нашим классам, также являются уникальными для них; все остальное они наследовали от класса Sprite.

Рассмотрим еще один эффект наследования, а именно возможность использовать один класс вместо другого. В примере 1.9 замените две ссылки, указывающие на класс Sprite, ссылками на класс MovieClip. Поскольку класс MovieClip наследует все от класса Sprite, приложение должно работать нормально, в чем несложно убедиться. Причина, по которой мы использовали Sprite вместо MovieClip, состоит в том, что нам не нужен никакой «дополнительный багаж», а только необходимый нам минимум. Если же вы замените Sprite на предыдущий класс, который он наследует, DisplayObjectContainer, приложение не будет работать. Это означает, что приложению требуется одно из свойств класса Sprite, которое *не было* унаследовано.

Увеличение числа байт: если в примере 1.9 вы замените классом MovieClip его родительский класс Sprite, вы обнаружите, что ваш *swf*-файл стал несколько больше, чем он был с классом Sprite (708 байт вместо 679). Эти 29 байт[2], вероятно, не «раздуют» вашу программу слишком сильно, но использование в шаблонах проектирования дополнительных классов, занимающих без надобности лишние байты, вполне может это сделать. (Когда вы получите контракт потому, что ваше приложение на 1 байт меньше, чем у конкурента, вы будете рады, что мы затронули данную тему.)

---

[1] Во Flash CS4 список собственных свойств объекта MovieClip и свойств некоторых родительских объектов был расширен. – *Прим. науч. ред.*

[2] Размеры файлов равны указанным только в случае компиляции в Flash CS3. При использовании более поздних версий Flash они будут другими. – *Прим. науч. ред.*

Кроме отслеживания цепочки наследования классов в прилагаемом к ActionScript 3.0 «Language Reference», возможно, вам также стоит обратить внимание на то, что находится в пакетах, которые вы импортируете. Если вы опишете импорт как flash.display.*, то вы заберете к себе все то многое, что есть в пакете display. Вот почему важно импортировать только то, что действительно требуется для вашей работы. Например, запись flash.display.Sprite или flash.display.Shape сделает импорт более понятным и более компактным.

Создание классов для наследования

Как легко увидеть из структуры наследования классов ActionScript 3.0, хорошо спланированная архитектура наследования классов очень сильно помогает при разработке сложного приложения. Чтобы показать, как работает наследование, изнутри, в следующем примере мы рассматриваем простую структуру классов для наших четвероногих друзей. Даже такой незамысловатый пример дает возможность понять, что наследуется от классов, а что реализуется в приложении.

Примеры с 1.13 по 1.16 посвящены приложению, разработка которого иллюстрирует принципы наследования. Первый класс, `QuadPets`, является родительским классом, или суперклассом для остальных классов и имеет конструктор, сигнализирующий с помощью функции `trace()` о каждом новом порождении объекта данного класса. Любой класс-потомок `QuadPets` наследует все его методы и интерфейсы.

Пример 1.13. QuadPets.as

```
package
{
    public class QuadPets
    {
        public function QuadPets():void
        {
            trace("QuadPets is instantiated");
        }
        public function makeSound():void
        {
            trace("Superclass:Pet Sound");
        }
    }
}
```

Любой класс, использующий выражение `extends` («расширяет») в своем описании, наследует характеристики того класса, который он расширяет (т. е. наследует). Класс, который расширяют, называется *суперклассом* (или родителем). Класс, который его расширяет, называют *подклассом*. Классы `Dog` и `Cat` являются подклассами класса `QuadPets`. Они наследуют всю функциональность этого суперкласса. Чтобы увидеть это воочию, нам потребуется сначала их создать.

Пример 1.14. Dog.as

```
package
{
    public class Dog extends QuadPets
    {
        public function Dog():void
        {
        }
        public function bark():void
        {
            trace("Dog class: Bow wow");
        }
    }
}
```

Пример 1.15. Cat.as

```
package
{
    public class Cat extends QuadPets
    {
        public function Cat():void
        {
        }
        public function meow():void
        {
            trace("Cat class: Meow");
        }
    }
}
```

Чтобы проверить тот факт, что классы `Dog` и `Cat` наследуют операции своего суперкласса, тестовый класс `TestPets` вызывает метод суперкласса `makeSound()`. Как легко видеть из определения классов `Dog` и `Cat`, такого метода в их конструкции нет, и, значит, он должен попадать к ним из класса `QuadPets`.

Пример 1.16. TestPets.as

```
package
{
    import flash.display.Sprite;

    public class TestPets extends Sprite
    {
        public function TestPets():void
        {
            var dog:Dog=new Dog();
            dog.makeSound();
            dog.bark();
            var cat:Cat=new Cat();
```

```
            cat.makeSound();
            cat.meow();
        }
    }
}
```

Помимо метода `makeSound()` у объектов классов `Dog` и `Cat` вызываются их собственные методы `bark()` и `meow()`. Кроме того, при запуске такого приложения вы увидите:

```
QuadPets is instantiated
```

Эта строчка выводится командой

```
trace("QuadPets is instantiated");
```

которая находится в конструкторе класса `QuadPets`. Она срабатывает всегда, какого бы класса объект ни создавался. Таким образом, подклассы не только приобретают возможность использовать любые методы своего суперкласса, но и наследуют все действия из его конструктора.

Откройте документ Flash и наберите `TestPets` в поле класс документа. Когда вы его запустите, вы должны получить следующий вывод на экран:

```
QuadPets is instantiated
Superclass:Pet Sound
Dog class: Bow wow
QuadPets is instantiated
Superclass:Pet Sound
Cat class: Meow
```

Из него видно, что объекты `Dog` и `Cat` показывают два сообщения от своего суперкласса (`QuadPets is instantiated, Superclass:Pet Sound`) и одно сообщение, уникальное для своего собственного класса (`Dog class: Bow wow, Cat class: Meow`). Эти примеры демонстрируют, как работает наследование. В практических приложениях и в шаблонах проектирования наследование классов планируется таким образом, чтобы оно исключало ненужное повторение операций при проектировании и помогало компоновать программу для достижения общей цели.

Использование интерфейсов и абстрактных классов в ActionScript 3.0

Наследование связано также с двумя другими структурами: *интерфейсами* и *абстрактными классами*. Однако связь наследования со структурой `interface` (ключевым словом в ActionScript 3.0) или абстрактным классом немного отличаются от его связи со структурой `class`, которую мы уже рассмотрели.

Конструкция interface

Прежде всего, в данном контексте слово `interface` означает ключевое слово в ActionScript 3.0, а не интерфейс классов, обсужденный нами в разделе «Инкапсуляция и шаблоны проектирования». Если класс можно в некотором смысле назвать абстракцией объекта, то `interface` можно назвать абстракцией методов. Структуры `interface` широко используются в шаблонах проектирования. Вы увидите несколько тому примеров в главах, начиная с 5-ой, посвященной шаблону Адаптер.

Чтобы понять, что делает `interface`, мы разберем простой пример его использования. Однако в чем состоит их польза, нам станет ясно позднее, при рассмотрении шаблонов проектирования. Пример 1.17 показывает, как создается типичный `interface`. Созданию полного приложения посвящены все примеры с 1.17 по 1.20.

Сначала мы создадим наш интерфейс. Как легко видеть из примера 1.17, его единственная функция проста и абстрактна. У нее есть собственное имя, параметр и возвращаемый тип, но, заметьте, она состоит только из одной строки своего описания.

Пример 1.17. BandFace.as

```
package
{
    // Интерфейс
    public interface BandFace
    {
        function playInstrument(strum:String):void;
    }
}
```

Каждая реализация интерфейса должна в точности повторять объявленную ранее структуру методов интерфейса; если это будет не так, вы получите сообщение об ошибке. Пока сигнатуры методов совпадают, все должно работать прекрасно. Пример 1.18 – наша первая реализация интерфейса `BandFace`.

Пример 1.18. Guitar.as

```
package
{
    public class Guitar implements BandFace
    {
        public function Guitar() {}

        public function playInstrument(strum:String):void
        {
            trace("Playing my air "+ strum);
        }
    }
}
```

Посмотрев на то, как в следующем примере 1.19 реализован класс Bongo, вы можете решить, что наш метод построен некорректно. Но, хотя он полностью отличается от класса Guitar в деталях, сигнатура этого метода точно такая же.

Пример 1.19. Bongo.as

```
package
{
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;

    public class Bongo implements BandFace
    {

        public function Bongo(){}

        private var sound:Sound;
        private var playNow:SoundChannel;
        private var doPlay:URLRequest;

        public function playInstrument(strum:String):void
        {
            sound=new Sound();

            doPlay=new URLRequest(strum);
            sound.load(doPlay);
            playNow=sound.play();
        }
    }
}
```

Работая с интерфейсом, не забывайте, что, сколько бы не было методов в нем, пока реализация этого интерфейса включает в себя каждый его метод и их сигнатуры совпадают, проблем возникнуть не должно.

У вас может возникнуть вопрос, причем тут наследование. Учитывая тот факт, что вам приходится реализовывать все указанные в интерфейсе методы, все это больше напоминает их модификацию, чем какое-то наследование. Однако все подклассы BandFace наследуют именно его интерфейс, а не реализацию.

В заключение, чтобы протестировать приложение, класс MakeSound, приведенный в примере 1.20, использует классы и их различные реализации единственного метода интерфейса BandFace. Вам дополнительно потребуется *mp3*-файл с именем *bongo.mp3* (используйте любой доступный вам *mp3*-файл), находящийся в том же каталоге, что и файл *MakeSound.as*.

Пример 1.20. MakeSound.as

```
 package
{
    import flash.display.Sprite;
    public class MakeSound extends Sprite
    {
        private var guitar:BandFace;
        private var bongo:BandFace;

        public function MakeSound():void
        {
            guitar=new Guitar();
            guitar.playInstrument("Gibson");

            bongo=new Bongo();
            bongo.playInstrument("bongo.mp3");
        }
    }
}
```

Обратите внимание, что оба объекта, `guitar` и `bongo`, объявлены представителями своего суперкласса `BandFace`, а не классов `Guitar` и `Bongo`. Такая практика следует главному принципу повторного использования в объектно-ориентированном программировании: *Используйте в программе интерфейсы, а не их реализации*.

Это делается для сохранения гибкости и возможности повторного использования свойств и методов. Более подробно данный принцип будет рассмотрен далее в этой главе, а также в главе 8, пока же просто возьмите его себе на заметку.

Несколько слов об используемых в данной книге правилах именования интерфейсов и абстрактных классов. Поскольку она посвящена в большей степени объектно-ориентированному программированию и шаблонам проектирования, мы используем такие правила именования, которые лучше отражают и объясняют структуру различных шаблонов проектирования. В результате мы не всегда следуем некоторым общепринятым соглашениям об именовании. Одно из таких соглашений предписывает начинать все имена интерфейсов с заглавной буквы `I`. По этому правилу интерфейс `BandFace` назывался бы `IBandFace`. Когда использование заглавной буквы `I` не мешает нам ясно описывать структуры шаблонов проектирования, мы ее используем. Но когда у нас в шаблоне имеется несколько интерфейсов, мы довольно часто не следуем данному правилу. Другое соглашение об именовании гласит, что все абстрактные классы необходимо именовать в виде `Abstract+Имя`. Тогда абстрактный класс `Horses` носил бы имя `AbstractHorses`. Как и в предыдущем случае, наше

стремление добиться ясного изложения структуры шаблонов привело нас к решению отказаться от использования этого соглашения. Мы выделяем абстрактные классы с помощью комментариев в коде программы. На протяжении всей книги мы стараемся максимально понятно именовать различные классы. Вы, возможно, захотите использовать вышеуказанные общие соглашения для поддержания ясности вашего кода, как только достаточно хорошо освоите их.

Абстрактные классы и изменение наследования

По мере вашего знакомства с шаблонами проектирования вы все чаще и чаще будете встречаться с интерфейсами и их ближайшими родственниками – *абстрактными классами*. В языке ActionScript 3.0 с абстрактными классами связана небольшая проблема, вызванная тем, что никакой класс нельзя объявить абстрактным явным образом. Тогда как для объявления класса публичным можно воспользоваться ключевым словом `public`, для абстрактных классов в языке ActionScript 3.0 (и в ECMAScript) аналогичная возможность отсутствует (в отличие, например, от языка Java).

Тем не менее создавать абстрактные классы в ActionScript 3.0 можно. Для этого нужно лишь создать обычный класс и рассматривать его как абстрактный. Как и интерфейсы, абстрактные классы могут содержать в себе методы, которые не полностью имплементированы. Точнее, абстрактные классы могут иметь подклассы, и тогда все абстрактные методы замещаются и реализуются подобно тому, как это происходит с методами при использовании интерфейсов. Однако абстрактные классы могут содержать в себе и полностью реализованные методы. Итак, если вам необходимо иметь одновременно абстрактные и конкретные методы, абстрактные классы предоставляют вам такую возможность; это используется нами в шаблонах Фабричный метод (глава 2), Декоратор (глава 4), Компоновщик (глава 6), а также в некоторых других.

Из определения наследования вы знаете, что когда один класс является подклассом другого, он наследует все его методы. Работая с абстрактными классами, вы не реализуете его, а создаете его подкласс и затем имплементируете этот подкласс и его методы. Ввиду того что, по крайней мере, несколько методов абстрактного класса в нем не реализованы, вы должны *заместить* их на свою собственную реализацию. Ключевое слово `override` («замещение») позволяет изменить метод унаследованного класса, добавив в него собственную реализацию, но сохранив его сигнатуру. Вам нужно проследить, чтобы имя, число и типы параметров замещающего метода полностью совпадали с его описанием в абстрактном классе. Другими словами, когда вы замещаете метод из абстрактного класса, вы рассматриваете метод точно так же, как метод `interface`.

В примерах с 1.21 по 1.23 рассматривается приложение, иллюстрирующее работу простого абстрактного класса. Абстрактный класс имеет два метода: один абстрактный и один конкретный.

Пример 1.21. AbstractClass.as

```
package
{
    // Абстрактный класс
    public class AbstractClass
    {
        function abstractMethod():void {}
        function concreteMethod():void
        {
            trace("I'm a concrete method from an abstract class")
        }
    }
}
```

Подкласс абстрактного класса наследует все его методы и интерфейс и имплементирует абстрактный метод путем его замещения. Однако этот подкласс оставляет конкретный метод в исходном виде и не пытается изменить его существующую в суперклассе реализацию.

Пример 1.22. Subclass.as

```
package
{
    // Подкласс абстрактного класса
    public class Subclass extends AbstractClass
    {
        override function abstractMethod():void
        {
            trace("This is the overidden abstract method");
        }
    }
}
```

Когда приложение в конечном счете реализует метод, объявленный в абстрактном классе, оно делает это путем описания объекта через его интерфейс (`AbstractClass`), а создает с помощью подкласса (`Subclass`). Итак, переменная `doDemo` описывается через наш первоначальный `AbstractClass`, а создается через его подкласс `Subclass`.

Пример 1.23. ImplementSub.as

```
package
{
    // Реализация подкласса абстрактного класса
    import flash.display.Sprite;
```

```
public class ImplementSub extends Sprite
{
    private var doDemo:AbstractClass;

    public function ImplementSub()
    {
        doDemo=new Subclass();
        doDemo.abstractMethod();
        doDemo.concreteMethod();
    }
}
```

Ниже показано, что появится на выходе, когда вы запустите это тестовое приложение:

```
This is the overidden abstract method
I'm a concrete method from an abstract class
```

Наверно, сейчас вы ломаете голову над вопросом, зачем вообще проходить через такие сложности, если все это же можно реализовать без абстрактных классов? Вместо того чтобы создавать подкласс абстрактного класса, замещать один из его методов и реализовывать его в подклассе, мы могли бы сразу написать оба метода так, как нам нужно, и ничего не замещать. Следующий раздел постарается ответить на данный вопрос.

Зачем использовать интерфейсы и абстрактные классы?

Чтобы понять, зачем нужны интерфейсы и абстрактные классы, мы должны вспомнить общую цель использования шаблонов проектирования. Она состоит в *возможности многократно использовать объектно-ориентированные компоненты программ*. В книге мы рассматриваем довольно простые примеры, чтобы лучше объяснить понятия. Однако типичные программы, а также их алгоритмы обычно гораздо сложнее. После того как вы закончите свой программный проект, вам может потребоваться внести в него изменения. Чем больше и сложнее проект, тем труднее повторно использовать ваши наработки, поддерживать связи между его компонентами и вообще менять в нем что-либо, не разрушая его и не внедряя в него код, из-за которого будущие изменения будут невозможны в принципе.

Чтобы проиллюстрировать все это, давайте еще раз посмотрим на наше приложение в примерах 1.21–1.23 из предыдущего раздела. Представьте, что сложность класса `AbstractClass` возросла раз в десять. Увеличьте во столько же и число его подклассов. Ситуация сильно усложняется. А теперь представьте, что вам надо изменить функциональность абстрактного метода в `AbstractClass`, но при этом сохранить все построенные интерфейсы и связи.

Если вы используете абстрактный класс, вы можете создать новый его подкласс, заместить его абстрактные методы и использовать их по-

своему. Чтобы посмотреть, как все это работает, мы создадим новый подкласс AbstractClass и изменим его абстрактный метод. Мы больше ничего не изменяем в нашем приложении, так что нам не нужно волноваться о том, как оно будет продолжать работать, поскольку мы можем отделить наш новый подкласс и его метод и реализовать их только там, где это действительно нам нужно. Примеры 1.24 и 1.25 показывают нам два новых класса, в которых мы хотим что-то изменить.

Пример 1.24. SubclassChange.as

```
package
{
    // Новый подкласс абстрактного класса с некоторыми изменениями
    public class SubclassChange extends AbstractClass
    {
        override function abstractMethod():void
        {
            trace("This is the new abstractMethod!!")
            trace("Made just one little important change.");
            trace("But this still works just fine!");
        }
    }
}
```

Обратите внимание, вместо одной функции trace() в примере 1.24 используются три такие функции, что имитирует случай усложненного алгоритма работы данного метода. Однако пример 1.25, который реализует конечное приложение, остается практически идентичным примеру 1.23.

Пример 1.25. ImplementSubChange.as

```
package
{
    // Реализация измененного подкласса абстрактного класса
    import flash.display.Sprite;

    public class ImplementSubChange extends Sprite
    {
        private var doDemo:AbstractClass;

        public function ImplementSubChange()
        {
            doDemo=new SubclassChange();
            doDemo.abstractMethod();
            doDemo.concreteMethod();
        }
    }
}
```

Все изменения скрываются в строке

```
doDemo.abstractMethod();
```

Хотя эта строка совпадает со строкой из примера 1.23, результат работы программы показывает, что в ней произошло довольно много изменений:

```
This is the new abstractMethod!!
Made just one little important change.
But this still works just fine!
I'm a concrete method from an abstract class
```

Однако важно *не* то, что произошли изменения. Важно то, что вы можете совершать подобные изменения, не нарушая работу других элементов вашего приложения. Таким образом, вы можете сохранять функциональность вашего приложения, производя в нем нужные вам замены, без необходимости полностью его переписывать.

Полиморфизм

В основе слова «полиморфизм» лежит другое слово, которое позволяет легко описать этот процесс, – «метаморфоза». Оно происходит от греческого *metamorphoun*, что означает «превращение». Одно из определений метаморфозы описывает этот процесс как чудесное преобразование, подобное тому, что может произвести волшебник. Таким образом, можно считать, что полиморфизм дает программисту силу волшебника.

Реализация полиморфизма с помощью абстрактных классов

Согласно другому определению полиморфизма, с его помощью можно создавать *много* («поли») *форм* («морф») чего-либо. В примерах с 1.21 по 1.25 `abstractMethod` имел более одной формы. Это простой случай полиморфизма. Чтобы увидеть его в более практическом приложении, давайте займемся различными музыкальными пристрастиями людей. Предположим, что вы создаете класс с методом, который описывает музыкальные вкусы конкретного человека, и хотите с помощью этого метода построить приложение для разных музыкальных жанров. Примеры с 1.26 по 1.31 содержат простой вариант такого приложения. Наш корневой класс будет называться `Polymorphism` в честь той концепции, которую он иллюстрирует.

Пример 1.26. Polymorphism.as

```
package
{
    // Абстрактный класс
    public class Polymorphism
    {
```

```
        public function myMusic():void
        {
            // Резервные детали для подклассов
        }
    }
```

Пример 1.27. Rock.as

```
package
{
    public class Rock extends Polymorphism
    {
        override public function myMusic():void
        {
            trace("Play Jimmie");
        }
    }
}
```

Пример 1.28. Classic.as

```
package
{
    public class Classic extends Polymorphism
    {
        override public function myMusic():void
        {
            trace("Play Mozart");
        }
    }
}
```

Пример 1.29. Country.as

```
package
{
    public class Country extends Polymorphism
    {
        override public function myMusic():void
        {
            trace("Play Willie");
        }
    }
}
```

Пример 1.30. Jazz.as

```
package
{
    public class Jazz extends Polymorphism
    {
        override public function myMusic():void
        {
            trace("Play Coltrane");
```

```
        }
    }
}
```

Пример 1.31. PlayMusic.as

```
package
{
    import flash.display.Sprite;
    public class PlayMusic extends Sprite
    {
        var rock:Polymorphism;
        var classic:Polymorphism;
        var country:Polymorphism;
        var jazz:Polymorphism;

        public function PlayMusic():void
        {
            rock=new Rock();
            rock.myMusic();
            classic=new Classic();
            classic.myMusic();
            country=new Country();
            country.myMusic();
            jazz=new Jazz();
            jazz.myMusic();
        }
    }
}
```

Когда вы протестируете эту программу, вы увидите следующее:

```
Play Jimmie
Play Mozart
Play Willie
Play Coltrane
```

Как видите, ничего особо сложного и покрытого мраком тайны здесь нет. Это просто полиморфизм. Все наши переменные изначально принадлежат абстрактному классу `Polymorphism`. Затем они реализуются с помощью различных его подклассов и запускают один и тот же метод `myMusic()`. Однако, хотя все они одного типа данных (или супертипа), `Polymorphism`, каждое использование этого метода порождает свой уникальный результат.

Глядя на эту программу и на собрание песен в вашем MP3-плеере, вы, возможно, думаете: «Здесь явно не все категории музыки». Где же R&B, альтернатива, кантри, хип-хоп и фолк-музыка? В этом мы не можем не согласиться с вами. Создайте, используя полиморфизм, новые подклассы, добавив все нужные вам категории. (Это упражнение не из тех, когда единственно правильный ответ есть в конце главы или книги. Полная свобода вашего творчества! Программисты не поймут полиморфизм,

пока не опробуют его в собственном коде. Проверьте, хорошо ли вы усвоили материал.) Кстати, обратите внимание на то, что при использовании полиморфизма вы можете делать все нужные вам изменения, ничего не меняя в других классах. В этом и состоит суть полиморфизма.

Реализация полиморфизма с помощью интерфейсов

Ключевое слово и структура `interface` языка ActionScript 3.0 является еще одним мощным средством полиморфизма для объектно-ориентированного программирования и шаблонов проектирования. Предположим, что вы создаете веб-сайт для электронной коммерции. Вы совершенно не представляете, сколько новых продуктов будет выставлено на этом сайте, но вы знаете, что их будет много и они будут регулярно меняться. Вам понадобится что-то, что поможет вам справиться с подобным разнообразием продуктов и сможет правильно отображать их на вашем сайте.

Вы знаете, что вам потребуется выполнять с продуктами следующие операции, но не уверены насчет деталей их реализации:

- Выводить их описание
- Выводить цену
- Выводить представление продукта

Одним из способов реализации подобного торгового сайта является создание класса, содержащего методы для выполнения всех трех указанных операций. Однако предположим, вы узнаете, что ваша клиентка желает, чтобы различные типы товаров отображались разными способами. Она желает использовать динамически загружаемые клипы для всей видео продукции (например, телевизоров), звуки для музыкальной продукции (такой как MP3-плееры) и графические изображения для всех прочих товаров. Операции вывода описания продукта и его цены будут достаточно стандартными для всех категорий продуктов, а операцию их отображения нужно сделать очень гибкой. Тут-то нам и пригодится полиморфизм. Мы могли бы использовать для нашей задачи абстрактный класс, но чтобы обеспечить еще большую гибкость на случай появления непредвиденных требований, мы используем интерфейсы.

Примеры с 1.32 по 1.36 составляют указанное приложение для электронной коммерции. Интерфейс `IBiz` содержит основные операции с уникальными сигнатурами, но без деталей их реализации. Это позволит нам создавать уникальные, нужные нам детали, пока мы сохраняем все сигнатуры прежними.

Пример 1.32. IBiz.as

```
package
{
    public interface IBiz
```

```
    {
        function productDescribe():String;
        function productPrice(price:Number):String;
        function productDisplay(product:String):void;
    }
}
```

Пример 1.33 показывает нам кое-что новенькое. Класс `Plasma` расширяет собой один класс, `Sprite`, и реализует другой, `IBiz`. Для проигрывания видеоклипов требуется функциональность объекта `Sprite`, но и методы из интерфейса `IBiz` нужны нам по-прежнему; поэтому мы используем оба с помощью ключевых слов `extends` и `implements`.

Пример 1.33. Plasma.as

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.display.Sprite;

    public class Plasma extends Sprite implements IBiz
    {

        private var ns:NetStream;
        private var vid:Video;
        private var priceNow:Number;

        public function productDescribe():String
        {
            return "42 inch TV with Plasma screen";
        }

        public function productPrice(price:Number):String
        {
            priceNow=price;
            return "$" + priceNow + "\n";
        }

        public function productDisplay(flv:String):void
        {
            var nc:NetConnection=new NetConnection();
            nc.connect(null);
            ns=new NetStream(nc);
            ns.play(flv);
            vid=new Video();
            vid.attachNetStream(ns);
            addChild(vid);
        }
    }
}
```

Сравнение примеров 1.33 и 1.34 показывает, что методы productDescribe() и productPrice() идентичны, за исключением возвращаемой строки в productDescribe(). На самом деле, добавив строковый параметр к методу productDescribe(), мы легко можем сделать их одинаковыми. Однако главная задача приложения – демонстрация полиморфизма, поэтому создание различных форм методов неизбежно.

Особенно явно полиморфизм проявляется в реализации метода productDisplay() в двух данных примерах. В примере 1.33 проигрывается видеоклип, а в примере 1.34 – звук. Обратите внимание, что одинаковые сигнатуры данного метода сохраняются в обоих этих примерах.

Пример 1.34. MP3Player.as

```
package
{
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;

    public class MP3Player implements IBiz
    {
        private var sound:Sound;
        private var playChannel:SoundChannel;
        private var doPlay:URLRequest;
        private var MP3Price:Number;

        public function productDescribe():String
        {
            return "MP3Player with 1 Terabyte of Memory";
        }

        public function productPrice(price:Number):String
        {
            MP3Price=price;
            return "$" + MP3Price + "\n";
        }

        public function productDisplay(song:String):void
        {
            sound=new Sound();
            playChannel=new SoundChannel();
            doPlay=new URLRequest(song);
            sound.load(doPlay);
            playChannel=sound.play();
        }
    }
}
```

Следующий наш пример, 1.35, также отличается в деталях реализации метода productDisplay(). Вместо операций, связанных с видео или

звуками, он содержит действия по загрузке файлов. Но, как и в предыдущих двух примерах, сигнатура этого метода и интерфейсы здесь сохраняются.

Пример 1.35. Computers.as

```
package
{
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.display.Sprite;

    public class Computers extends Sprite implements IBiz
    {
        private var hotz:Number;
        private var loadPix:Loader;
        private var namePix:URLRequest;

        public function productDescribe():String
        {
           return "New 10 gHz processor, 30 gigabyte RAM, includesv 32 inch screen";
        }

        public function productPrice(price:Number):String
        {
            hotz=price;
            return "$" + hotz + "\n";
        }

        public function productDisplay(computer:String):void
        {
            loadPix=new Loader();
            namePix=new URLRequest(computer);
            loadPix.load(namePix);
            addChild(loadPix);
        }
    }
}
```

Теперь, когда все нужные нам классы и интерфейс реализованы, нам осталось лишь создать класс для тестового приложения. Как вы уже знаете, хороший тон объектно-ориентированного программирования и подготовка к использованию шаблонов проектирования предполагают использование в программе только интерфейса, а не его реализации, как мы и делали. Однако если вы посмотрите на пример 1.36, вы увидите, что все типы данных там установлены в `Plasma`, `MP3Player` и `Computers`, а вовсе не `IBiz` интерфейс. Это произошло потому, что каждая реализация расширяет собой класс `Sprite`. Таким образом, у нас имеются два супертипа данных – `Sprite` и `IBiz`. В тестовом приложении все объекты объявлены типами конкретных реализаций, а не интерфейсом. В разделе «Исполь-

зование сложных интерфейсов» позднее в данной главе вы узнаете, как можно конструировать ваши интерфейсы и их реализации так, чтобы в описании можно было использовать интерфейс даже при наличии нескольких типов в его реализации. (В примере 1.20 вы уже видели, как при описании объектов использовался интерфейс, а не его реализация.)

Пример 1.36. DoBusiness.as

```
package
{
    import flash.display.Sprite;

    public class DoBusiness extends Sprite
    {
        public function DoBusiness()
        {
            // Телевизор
            var tv:Plasma=new Plasma();
            trace(tv.productDescribe());
            trace(tv.productPrice(855));
            tv.productDisplay("plasma.flv");
            tv.x=160;
            tv.y=110;
            addChild(tv);

            // MP3-плеер
            var mp3:MP3Player=new MP3Player();
            trace(mp3.productDescribe());
            trace(mp3.productPrice(245));
            mp3.productDisplay("bongo.mp3");

            // Компьютеры
            var computers:Computers=new Computers();
            trace(computers.productDescribe());
            trace(computers.productPrice(1200));
            computers.productDisplay("whizbang.gif");
            addChild(computers);
        }
    }
}
```

Для работы данного примера вам потребуются *flv*-файл, *mp3*-файл и *gif*-файл. Назовите *flv*-файл *plasma.flv*, *mp3*-файл – *bongo.mp3*, а *gif*-файл – *whizbang.gif*. Содержимое этих файлов не существенно, но имена и их типы важны.

Откройте новый документ Flash, сохраните его в том же каталоге, что и остальное приложение, и введите `DoBusiness` в поле класса документа. Когда вы его протестируете, окно сообщений будет выглядеть следующим образом:

```
42 inch TV with Plasma screen
$855

MP3Player with 1 Terabyte of Memory
$245

New 10 gHz processor, 30 gigabyte RAM, includes 32 inch flat screen monitor
$1200
```

Помимо этих сообщений вы должны увидеть нечто вроде изображенного на рис. 1.6.

Рис. 1.6. *Отображение графического файла и видео*

Кроме того, вы должны услышать звук проигрываемого *mp3*-файла. (И все это мультимедийное многообразие стало возможным благодаря использованию полиморфизма.)

Принципы разработки с помощью шаблонов проектирования

Базовые принципы применения шаблонов проектирования были заложены «бандой четырех» в их классической книге «Design Patterns: Elements of Reusable Object-Oriented Software». Вот эти два принципа:

- Работать с интерфейсами, а не с их реализациями
- Композиция объектов предпочтительнее наследования классов

Если вы пропустили раздел, посвященный четырем уже знакомым вам основным принципам ООП, то в нем мы уже немного обсудили первый принцип применения шаблонов проектирования. Но не волнуйтесь, мы рассмотрим его здесь еще раз и более подробно.

Прежде чем двигаться в данном направлении, нам надо прояснить ряд базовых терминов. Хотя все они довольно простые, отсутствие полного понимания их использования в наших специфических контекстах может сильно затруднить ваш путь. Мы будем активно работать с ними в последующих главах, поэтому очень важно разобраться с ними с самого начала.

Имплементация

В целом термин *имплементация* обозначает фактическую реализацию чего-либо в коде программы. Поэтому, когда мы будем говорить об имплементации, мы будем обращаться к реальному коду программы. Имплементация может описывать как фактическую реализацию программы, так и реализацию отдельных методов и классов.

У вас могут возникнуть затруднения, когда ключевое слово `implements` будет использоваться в сочетании со структурой `interface`. Например, строка

```
class MyClass implements IMyInterface
```

передает *интерфейс* `IMyInterface` классу `MyClass`. В данном случае фраза, что `MyClass` *имплементирует* интерфейс `IMyInterface`, означает, что в `MyClass` будут использованы все сигнатуры, описанные в `IMyInterface`, и к ним будут добавлены некоторые их реализации. То есть класс `MyClass` возьмет из интерфейса `IMyInterface` все его методы и добавит к ним необходимый для их реальной работы код, сохраняя все их сигнатуры.

Один из двух главных принципов программирования с помощью шаблонов проектирования гласит: *используйте интерфейсы, а не их имплементацию*. Однако вы должны помнить, что слово *имплементация* может использоваться в разных контекстах и его специфическое значение сильно от них зависит. В этой книге вы увидите многократные упоминания имплементации в связи с различными шаблонами проектирования, и вам нужно будет учитывать прочие рассматриваемые элементы при ее обсуждении.

Состояние

Хотя в лексиконе ActionScript 3.0 нет такого ключевого слова, как *состояние* (в отличие, например, от слова `implements`), этот термин активно применяется при обсуждении шаблонов проектирования. По су-

ществу термин «состояние» используется при описании текущего состояния объекта. Большей частью вы увидите, что этот термин используется для описания значения важной переменной. Представьте себе, например, класс с одним методом, в котором устанавливается значение некоторой булевской переменной. Она может принимать значение «истина» либо «ложь». Следовательно, ее состояние может быть истинно либо ложно, что может обозначать «включено/выключено», «разрешено/запрещено» или любое другое бинарное состояние. Точный смысл значений «истинно» и «ложно» зависит от положения класса в шаблоне проектирования. Более специфическим примером является объект, который может проигрывать *mp3*-файл. Если он его проигрывает, то он в состоянии «проигрывание», а если нет, то – в состоянии «стоп».

Термин «состояние» используется также в контексте абстрактных автоматов в шаблоне проектирования Состояние, который будет рассматриваться нами в главе 10. При этом термин *сеть состояний* означает структуру данных, представляющую собой некоторую сеть состояний, переходы между которыми изменяют состояние всего приложения. Шаблон проектирования Состояние основан на объекте, поведение которого изменяется в зависимости от изменения его внутреннего состояния. Термин «состояние» представляется нам здесь более контекстуально-богатым, поскольку он является ключевой концепцией, вокруг которой строится весь этот шаблон проектирования.

Хотя в большей части случаев, когда вы видите слово «состояние», оно обычно обозначает всего лишь набор значений внутренних переменных объекта.

Клиент и запрос

В нашу эпоху Интернета и Всемирной паутины термин *клиент* используется, чтобы отличить источник запроса от *сервера*, к которому этот запрос обращен. Мы всегда рассматриваем клиента и сервер в паре. Термин *запрос* используется для обозначения того, что веб-страница была вызвана с сервера.

В контексте шаблонов проектирования говорят не о паре клиент/сервер, а о паре *клиент/объект*. А запрос – это то, что клиент посылает объекту, чтобы заставить последнего выполнить какую-то операцию, то есть запустить какой-то его метод. В данном контексте *запрос является единственным способом заставить объект выполнить какой-то из своих методов*. Только запрос может запустить выполнение некоторой операции, поскольку другой доступ к его внутренним переменным закрыт согласно правилу инкапсуляции.

Но не забывайте о Flash Media Server-клиентах и серверах! Если вы работаете с FMS 2, вы должны помнить о его приложениях на стороне клиента и на стороне сервера. Вы можете вызвать метод на стороне сервера, инициировав запрос от объекта, находяще-

гося в клиентском приложении. Однако вы можете также вызвать клиентскую операцию запросом со стороны клиента и операцию на стороне сервера запросом со стороны серверного приложения. Так что если вы используете FMS 2, вам просто придется мысленно разделять все вышеназванные концепции.

В двух словах, *клиент* является источником *запроса* к методу объекта. Короткий пример покажет вам, как это выглядит на практике. В примерах 1.37 и 1.38 приводится приложение, которое ничего не делает, а только демонстрирует, что клиент обратился к объекту с запросом.

Пример 1.37. MyObject.as

```
package
{
    public class MyObject
    {
        private var fire:String;

        public function MyObject():void {}
        public function worksForRequest():void
        {
            fire="This was requested by a client";
            trace(fire);
        }
    }
}
```

Единственным методом класса `MyObject` является `worksForRequest()` («работает по запросу»). Данный метод инкапсулирован, поэтому существует только один способ выполнить его операции. К счастью, для этого достаточно всего лишь создать объект данного класса и добавить через точку имя метода к его имени, как это показано в примере 1.38.

Пример 1.38. MyClient.as

```
package
{
    import flash.display.Sprite;

    public class MyClient extends Sprite
    {
        var myClient:MyObject;
        public function MyClient():void
        {
            myClient=new MyObject();
            myClient.worksForRequest();
        }
    }
}
```

В итоге мы получим:

```
This was requested by a client
```

Под клиентом в данном случае подразумевается объект, чей метод был вызван нашим приложением. Все взаимодействия клиент/объект происходят только через запросы. Часто клиент добавляет в запрос некоторые специфические детали в виде параметров, но концепция запроса от этого не меняется и представляет собой не более чем вызов метода у объекта, владеющего этим методом.

Использование интерфейсов, а не их имплементаций

Чтобы разобраться, почему авторы шаблонов проектирования призывают использовать при программировании интерфейсы, а не их имплементации, вам нужно понять, во-первых, общую цель построения гибкого и поддающегося повторному использованию кода программ, а во-вторых, сложности, связанные с управлением зависимостями в больших программах.

Как мы уже говорили в этой главе, общей целью применения шаблонов проектирования является создание многократно используемого кода. Для осуществления этой задачи код должен быть гибким. Гибкость кода *не* означает, что ваши приложения будут работать или компилироваться быстрее. Но она поможет вам создавать код, который можно будет использовать в других проектах. Чем больше времени и сил вы тратите на создание вашей программы, чем больше разработчиков трудится над проектом, тем важнее становится данная цель.

Управление зависимостями

Потребность в большей гибкости кода влечет за собой необходимость эффективно управлять зависимостями в наших приложениях. Если ваш код зависит от конкретной имплементации чего-либо и эта имплементация меняется (по каким-либо причинам), то данная зависимость может привести либо к неожиданным результатам работы приложения, либо вообще к его отказу работать. При зависимости только от интерфейсов ваш код отделен от их имплементации, что позволяет последней изменяться. Это не значит, что вы совсем избавляетесь от зависимостей, просто вы начинаете управлять ими более гибко. Ключевым элементом в этом процессе является отделение проектирования приложения от его имплементации. При этом клиент также отделяется от имплементации.

Как мы уже говорили, в языке ActionScript 3.0 вы можете применять конструкцию `interface` или абстрактные классы для реализации подобной изменяемой зависимости. Однако, что бы вы из них ни использовали, вы должны понимать, каким способом происходит такое управ-

ление. Если вы будете применять конструкцию `interface`, то любое изменение в ней будет приводить к ошибке во всех использующих его клиентах. Например, предположим, что в вашей конструкции `interface` указаны пять различных методов. А вы решили, что вам требуется добавить туда еще два метода. Как только вы это сделаете, у всех ваших клиентов возникнут проблемы. Поэтому если вы пользуетесь в своей программе конструкцией `interface`, то вы должны сохранять объявленные с ее помощью интерфейсы в строго неизменном виде. А если вам потребуется некоторая новая функциональность, то просто создайте новый дополнительный интерфейс.

Альтернативой к использованию интерфейсов является использование абстрактных классов. Хотя конструкции для абстрактных классов не поддерживаются в ActionScript 3.0 так явно, как интерфейсы, вы легко можете создать обычный класс и делать с ним все, что вам нужно сделать с абстрактным классом. Как мы уже видели в нескольких примерах данной главы, начиная с примера 1.3, создание и использование абстрактных классов представляет собой просто следование обязательным для таких классов правилам. Например, такому: абстрактные классы никогда не бывают полностью имплементированными.

Преимуществом использования абстрактных классов перед использованием интерфейсов является то, что вы не нарушите работу ваших клиентов, если добавите новые методы в ваш исходный класс. Все абстрактные функции подобных классов должны замещаться и использоваться независимо друг от друга; однако если ваш клиент не будет никак применять новый метод, то последний будет им унаследован, но не будет в нем замещен и останется незадействованным. Что же касается конструкции `interface`, то, как вы знаете, каждый из указанных в ней методов обязательно должен быть имплементирован.

Другое преимущество использования абстрактных классов состоит в том, что вы можете установить поведение по умолчанию или даже имплементировать конкретные методы, которые будут унаследованы всеми его подклассами. Однако клиенту такая реализация может быть вовсе не нужна, и ее наличие приведет к непредсказуемым или нежелательным последствиям в его работе. В любом случае управление зависимостями при использовании интерфейсов и абстрактных классов проще, чем при использовании наследования в обычных классах и их методов.

Решение о том, что использовать, интерфейсы или абстрактные классы, зависит только от ваших требований к структуре зависимостей в вашем приложении. Если для вас важнее возможность легко добавлять новую функциональность к вашим объектам, лучше предпочесть абстрактные классы. А если вы хотите большей независимости от исходных классов, используйте интерфейсы. Что бы вы ни делали в вашем приложении, нужно всегда мыслить в перспективе. Если вы будете строить свою программу с прицелом на ее следующие версии и все

возможные пути ее развития и изменения, вам будет легче выбирать шаблоны проектирования для выполнения ваших задач.

Использование сложных интерфейсов

В примере 1.36 мы были вынуждены использовать в нашей программе конечные имплементации, а не их интерфейсы, как делалось в примере 1.20. Это произошло потому, что некоторые нужные нам методы принадлежали классам, которые одновременно являлись имплементацией интерфейса и расширяли собой другой класс. Поскольку такая проблема характерна для использования различных отображаемых объектов, она часто встречается при программировании на Flash и ActionScript 3.0. К счастью, для нее существует решение. (Оно представляет собой некоторый *искусственный прием* использования конструкций ActionScript 3.0. Тем не менее с его помощью вы можете создавать методы, которые нуждаются в некоторой структуре `DisplayObject` и запрограммированы на интерфейсы, а не на их имплементации.)

Если какой-то интерфейс включает в себя метод, которому требуется тип `DisplayObject`, то последний может быть частью этого интерфейса. Поскольку `Sprite` – это подкласс `DisplayObject`, его включение в интерфейс позволяет вам использовать при объявлении объектов интерфейс в случае, когда создаваемые вами классы является подклассами `Sprite` (или любого другого подкласса `DisplayObject`, такого как `MovieClip`.)

Чтобы вы могли посмотреть, как это все работает, мы написали приложение, создающее простой видеопроигрыватель (примеры 1.39 и 1.40). Класс `VidPlayer` реализует в себе все необходимое для работы такого видеопроигрывателя. Для этого он должен быть подклассом класса `Sprite`. Благодаря помещению метода-получателя типа `DisplayObject` в интерфейс `IVid`, появляется возможность использовать в программе этот интерфейс.

Пример 1.39. IVid.as

```
package
{
    import flash.display.DisplayObject;

    public interface IVid
    {
        function playVid(flv:String):void;
        function get displayObject():DisplayObject;
    }
}
```

В имплементации интерфейса `IVid` присутствует один важный элемент. При построении класса `VidPlayer` функция `displayObject()` реализуется через класс `DisplayObject`.

Пример 1.40. VidPlayer.as

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.display.Sprite;
    import flash.display.DisplayObject;

    public class VidPlayer extends Sprite implements IVid
    {

        private var ns:NetStream;
        private var vid:Video;
        private var nc:NetConnection;

        public function get displayObject():DisplayObject
        {
            return this;
        }

        public function playVid(flv:String):void
        {
            nc=new NetConnection();
            nc.connect(null);
            ns=new NetStream(nc);
            ns.play(flv);
            vid=new Video();
            vid.attachNetStream(ns);
            addChild(vid);
        }
    }
}
```

Не стоит забывать, что метод-получатель, использующий ключевое слово `get`, выглядит как свойство и обрабатывается, как оно. Любое обращение к `displayObject` является на самом деле запросом на вызов его метода. Хитрость в том, чтобы добавить объект в список отображения и одновременно с этим объявить его тип `DisplayObject`. Пример 1.41 делает именно это.

Пример 1.41. DoVid.as

```
package
{
    import flash.display.Sprite;

    public class DoVid extends Sprite
    {
```

```
            // Декларация типа через интерфейс
            private var showTime:IVid;

            public function DoVid()
            {
                // Проигрываем видео
                showTime=new VidPlayer();
                showTime.playVid("iVid.flv");
                // Подключаем экземпляр DisplayObject
                addChild(showTime.displayObject);
                showTime.displayObject.x=100;
                showTime.displayObject.y=50;
            }
        }
    }
```

Рассмотрим, как работает данный процесс. Первым делом объявляется объект `showTime`, типом которого является интерфейс `IVid`. Далее `showTime` присваивается объект класса `VidPlayer`, в котором есть все необходимое для проигрывания видео. Таким образом, он наследует одновременно и свойства класса `Sprite`, и интерфейс `IVid`. Потом клиент `showTime` проигрывает видео, используя метод `playVid()`. И, наконец, когда `showTime` добавляется в список отображения командой `addChild`, он добавляется и как потомок класса `VidPlayer`, и как потомок класса `DisplayObject` с помощью метода-получателя `displayObject`. Поскольку метод-получатель `displayObject` включен в список отображения, вы *не* получите следующего сообщения об ошибке:

```
1067: Implicit coercion of a value of type IVid to an unrelated type
    flash.display:DisplayObject.[1]
```

Интерфейс здесь указан потому, что он является типом нашего объекта, а не имплементации этого интерфейса. Включение в интерфейс метода-получателя типа `DisplayObject` позволяет избежать данной ошибки.

Преимущество композиции

В этой главе мы уже обсудили принцип программирования с использованием интерфейсов, а не их реализации. Второй принцип объектно-ориентированного программирования, постулированный Гаммой, Хелмом, Джонсоном и Влиссидесом («бандой четырех»), гласит: *композиция объектов предпочтительнее наследования классов*.

Чтобы понять второй ключевой принцип, нам нужно выяснить, что означает термин *композиция* и в чем его преимущества перед наследованием. Ведь этот принцип по существу утверждает, что ваши програм-

[1] Неявное приведение значения типа IVid к несвязанному типу `flash.display:DisplayObject`. — *Прим. науч. ред.*

мы будут лучше, если вы будете использовать в них композицию, а не наследование.

Означает ли это, что мы должны вообще отказаться от наследования? В конце концов оно является одной из основополагающих концепций объектно-ориентированного программирования. На самом деле «банда четырех» просто указала на то, что большинство программистов слишком сильно полагаются на наследование и должны больше использовать его в связке с композицией. Разработчики создают новые объекты из старых с помощью наследования, и композиция позволяет использовать старые и новые объекты вместе.

Лучший способ понять композицию – рассмотреть примеры ее использования. В этой книге шаблоны проектирования Состояние и Стратегия построены с применением композиции и делегирования полномочий. В шаблоне Состояние объект делегирует запросы объекту, представляющему текущее состояние в приложении. В шаблоне проектирования Стратегия специфические запросы делегируются объекту, представляющему стратегии (алгоритмы) решения проблем. Большим преимуществом обоих этих шаблонов является то, что они являются очень гибкими и не перестают работать в немодифицируемых зависимостях.

Создание композиций

Изучение композиции мы начнем с очень простого примера. В нижеследующем простом приложении вы увидите, как композиция и наследование используются вместе. Каждый из примеров будет демонстрировать одно из следующих отношений объектов:

- Отношение «является»: наследование объектов
- Отношение «имеет»: композиция объектов
- Отношение «использует»: один объект использует другой объект (без композиции или наследования)

В следующем разделе мы будем подробнее рассматривать эти отношения. А пока каждый из классов нашего приложения, приведенного в примерах с 1.42 по 1.44, просто демонстрирует одно из указанных выше отношений. Комментарии к примерам указывают тип отношения. Прежде всего нам требуется базовый класс, который мы будем потом делегировать.

Пример 1.42. BaseClass.as

```
package
{
    public class BaseClass
    {
        public function homeBase( )
        {
            trace("This is from the Base Class");
```

```
        }
    }
}
```

Композиция включает в себя ссылку на другой класс в определении своего класса. Пример 1.43 показывает нам, как в классе может быть использована композиция. Строка

```
private var baseClass:BaseClass;
```

содержит в себе ссылку на базовый класс при определении нового класса. Такая строка является основой для композиции. Теперь класс `HasBase` *имеет* в себе класс `BaseClass`. В данной конкретной имплементации класс `HasBase` создает представителя класса `BaseClass` в своем конструкторе. И наконец, общедоступная функция `doBase()` делегирует выполнение работы обратно в класс `BaseClass`.

Пример 1.43. HasBase.as

```
package
{
    // Композиция
    public class HasBase
    {
        private var baseClass:BaseClass;

        public function HasBase( )
        {
            baseClass=new BaseClass( );
        }
        public function doBase( )
        {
            baseClass.homeBase( );
        }
    }
}
```

В примере 1.44 класс `HasBase` используется для делегирования выполнения операции обратно в класс `BaseClass`. Все это делается без наследования свойств `BaseClass`. Хотя класс `HasBase` имеет внутри себя класс `BaseClass`, он не является наследником данного класса.

Пример 1.44. DoHasBase.as

```
package
{
    // Выполняет класс-композицию HasBase
    import flash.display.Sprite

    public class DoHasBase extends Sprite
    {
        private var hasBase:HasBase;
```

```
        public function DoHasBase()
        {
            hasBase=new HasBase();
            hasBase.doBase();
        }
    }
}
```

Конечно, в таких простых примерах трудно увидеть преимущества использования композиции перед наследованием. Позднее в этой книге при рассмотрении различных шаблонов проектирования преимущества проявятся яснее. А когда композиция применяется в больших проектах, особенно использующих шаблоны проектирования, ее преимущества становятся еще более очевидными. Вы действительно увидите, почему композиция предпочтительнее наследования, когда вам потребуется внести изменения в большой сложный проект, написанный совместно с несколькими разработчиками.

Использование делегирования

Делегирование – одна из самых важных концепций при работе с шаблонами проектирования, поскольку ее применение позволяет через композицию получить такие же возможности повторного использования, что и через наследование, но с гораздо большей гибкостью кода. Хотя делегирование обычно имеет дело с наследованием, *не* следует думать, что они неразрывно связаны друг с другом. Для нас важно понять их различие и то, как они работают вместе, потому что именно это обычно используется в шаблонах проектирования.

Чтобы показать отличия наследования от композиции, мы описывали отношения компонент в некотором приложении. Если класс Б является подклассом класса А, класс Б при этом также *является* классом А. Однако, если класс Б делегирует некоторые действия классу А с использованием композиции, тогда говорят, что класс Б *содержит* класс А.

Предположим, что у вас есть класс, способный загружать *swf*-файлы в нужной конфигурации. Вам нужна такая его функциональность, но еще вам хочется иметь возможность проигрывать *mp3*-файлы. Используя композицию, вы можете просто создать класс, который будет делегировать функции соответствующим классам. Рисунок 1.7 демонстрирует такие взаимоотношения между классами:

В подобной ситуации наследование оказывается мало полезным. Вы можете наследовать один класс, но не оба. Конечно, вы можете создать представителя каждого класса в третьем классе. Или вы можете просто создать подкласс одного класса и затем создать в нем представителя второго. Это будет новый класс, который «является» первым классом и «использует» второй класс для получения недостающей функциональности. Считается, что один класс «использует» другой класс, если первый содержит представителя второго, но не является вторым классом.

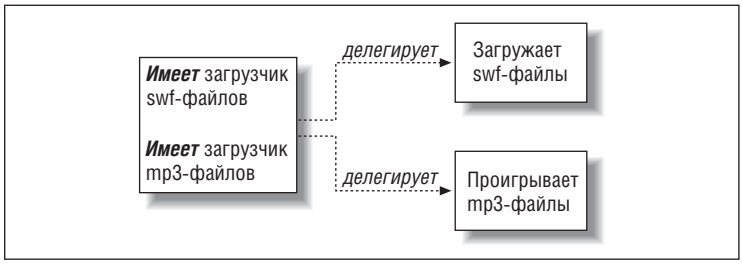


Рис. 1.7. *Делегирование различным классам*

Чтобы понять и лучше применять композицию, нам необходимо разобраться, как работает делегирование, и для этого будет полезен следующий, немного более реалистичный пример. Наша программа будет представлять собой мультимедийное приложение, которое способно проигрывать и записывать различные медиафайлы и использует в своей работе Flash и Flash Media Server 2 (FMS2). С помощью FMS2 можно проигрывать и *flv*-, и *mp3*-файлы, но записывать можно только *flv*-файлы. Это приложение создано с целью добавления в него дополнительной функциональности в будущем и демонстрации управления зависимостями.

Если вы уверены, что ваш медиасервер и ваше приложение в будущем обязательно изменятся, то вам следует минимизировать в нем все зависимости, используя композицию и делегирование. Например, предположим, что в будущем FMS2 изменится так, что станет возможно записывать *mp3*-файлы. Тогда вам придется лишь поменять класс `RecordAudio`, чтобы он мог записывать звуки. При этом ничто другое в приложении не будет затронуто. Или, например, у вас появится проигрыватель голографических файлов, который вы захотите добавить в ваше приложение. Вы сможете легко добавить еще один класс, который будет проигрывать и/или записывать голографические изображения, без каких-либо изменений в остальных частях приложения.

В примерах с 1.45 по 1.54 приводится наше приложение, представляющее типичным случай использования композиции. Сохраните все *.as*-файлы в одном каталоге.

Пример 1.45. Media.as

```
package
{
    // Абстрактный класс
    class Media
    {
        // Композиция: ссылки на два интерфейса
        var playMedia:PlayMedia;
        var recordMedia:RecordMedia;

        public function Media() {}
```

```
        public function doPlayMedia():void
        {
            // Делегирует функциональность классу PlayMedia
            playMedia.playNow();
        }
        public function doRecordMedia():void
        {
            // Делегирует функциональность классу RecordMedia
            recordMedia.recordNow();
        }
    }
}
```

Пример 1.46. VideoFlash.as

```
package
{
    // Конкретный подкласс Media: Video
    class VideoFlash extends Media
    {
        public function VideoFlash()
        {
            // Наследует ссылки композиции суперкласса
            playMedia = new PlayVideo();
            recordMedia = new RecordVideo();
        }
    }
}
```

Пример 1.47. Mp3.as

```
package
{
    // Конкретный подкласс Media: Audio
    public class Mp3 extends Media
    {
        public function Mp3()
        {
            // Наследует ссылки композиции суперкласса
            playMedia = new PlayAudio();
            recordMedia = new RecordAudio();
        }
    }
}
```

Пример 1.48. PlayMedia.as

```
package
{
    // Интерфейс для проигрывания
    interface PlayMedia
    {
        function playNow():void;
```

```
    }
}
```

Пример 1.49. PlayMedia.as

```
package
{
    // Конкретный подкласс проигрывателя PlayMedia: PlayVideo
    class PlayVideo implements PlayMedia
    {
        public function playNow():void
        {
            trace("Playing my video. Look at that!");
        }
    }
}
```

Пример 1.50. PlayAudio.as

```
package
{
    // Конкретный подкласс проигрывателя PlayMedia: PlayAudio
    class PlayAudio implements PlayMedia
    {
        public function playNow():void
        {
            trace("My MP3 is cranking out great music!");
        }
    }
}
```

Пример 1.51. RecordMedia.as

```
package
{
    // Интерфейс для записи
    interface RecordMedia
    {
        function recordNow():void;
    }
}
```

Пример 1.52. RecordVideo.as

```
package
{
    // Конкретный подкласс для записи RecordMedia: RecordVideo
    class RecordVideo implements RecordMedia
    {
        public function recordNow():void
        {
            trace("I'm recording this tornado live! Holy....crackle, crackle\n");
        }
    }
}
```

Пример 1.53. RecordAudio.as

```
package
{
    // Конкретный подкласс для записи RecordMedia: RecordAudio
    class RecordAudio implements RecordMedia
    {
        public function recordNow():void
        {
            trace("Rats! I can't record MP3 by itself.\n");
        }
    }
}
```

Пример 1.54. TestMedia.as

```
package
{
    import flash.display.Sprite;

    public class TestMedia extends Sprite
    {
        public function TestMedia()
        {
            var delVideo:Media=new VideoFlash();
            delVideo.doPlayMedia();
            delVideo.doRecordMedia();

            var delAudio:Media = new Mp3();
            delAudio.doPlayMedia();
            delAudio.doRecordMedia();
        }
    }
}
```

Создайте новый документ Flash, сохраните его в том же каталоге, что и файлы классов, и в поле класс документа наберите `TestMedia`. Когда вы его запустите, в выходном окне появятся сообщения о его работе.

```
Playing my video. Look at that!
I'm recording this tornado live! Holy....crackle, crackle

My MP3 is cranking out great music!
Rats! I can't record MP3 by itself.
```

Конечно, алгоритмы, которые вы будете применять в реальном приложении, будут гораздо сложнее, чем использование простой функции `trace`. Однако как бы сложны они ни были, зависимости в данной программе построены таким образом, что изменение одного из алгоритмов затронет те ее элементы, которые вы захотите изменить, но не коснется остальных.

Совместное использование композиции, наследования и представителей классов

Мы специально не проводили сравнительного анализа преимуществ и недостатков композиции и наследования, поскольку обычно композиция и наследование используются совместно. Аналогичная ситуация и с использованием представителей других классов. Под своей рекомендацией «композиция предпочтительнее наследования» «банда четырех» никогда не подразумевала идею о том, что следует работать с чем-то одним и исключать все остальное. Конечно, то, как эта рекомендация звучит, наводит на подобную мысль. Однако при объяснении данного принципа основатели шаблонов проектирования не только указывают, что наследование и композиция прекрасно работают вместе, но и подтверждают это собственными шаблонами проектирования.

Приложение в примерах с 1.45 по 1.54 иллюстрировало применение композиции. В нем показывались также наследование и использование представителей других классов в действии. Чтобы лучше понять отношения классов, посмотрите на рис. 1.8.

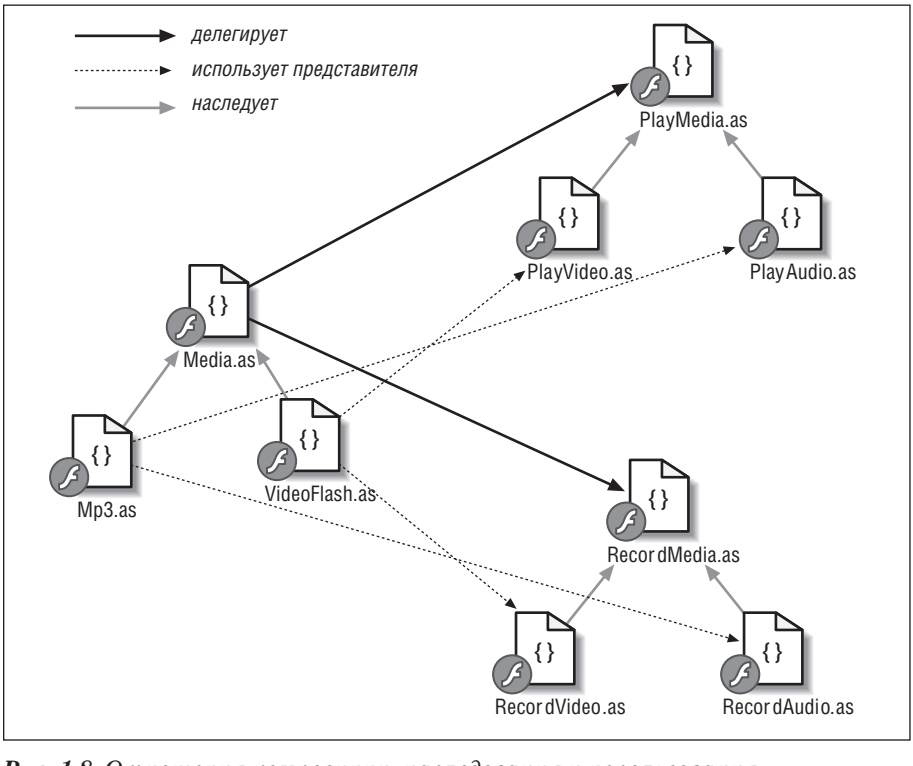


***Рис. 1.8.** Отношения композиции, наследования и использования представителя*

На нем видно, что класс Media *делегирует* операции классу RecordMedia. Это происходит с помощью ссылки на этот класс в его определении (см. пример 1.45).

```
var recordMedia:RecordMedia;
```

Вы можете также увидеть, что класс VideoFlash *наследует* свойства класса Media. И в то же время он *использует представителя* RecordMedia.

```
recordMedia = new RecordVideo( );
```

В свою очередь, RecordVideo *наследует* класс RecordMedia. И в этой точке мы опять возвращаемся к классу (RecordMedia), которому класс Media делегировал свои полномочия.

Трудно представить использование композиции без наследования в большинстве практических приложений. Поэтому, не проводя никакого сравнения их преимуществ, просто считайте их отныне одной командой. В главе 11 мы снова вернемся к теме предпочтения композиции перед наследованием в контексте использования композиции в шаблонах проектирования.

Планирование поддержки и развития приложений

Несколько лет назад мы создавали веб-сайт, которому суждено было со временем изменяться. В те времена никто еще не думал о шаблонах проектирования, но уже тогда из своего опыта мы знали, что сайт будет изменяться, и хотели быть уверены, что эти изменения будут возможны без полного переписывания его заново.

В том случае наш план сработал даже слишком хорошо. Сайт было легко обновлять, и в результате мы так и не озаботились его переделкой с целью реализации в нем новых появившихся концепций. Он начал выглядеть немного старомодно, и мы захотели провести обновление версии Flash, чтобы была возможность оптимизировать работу видео и использовать все новые возможности Flash CS3 и ActionScript 3.0. Вся эта история наглядно иллюстрирует нам один простой факт о разработке программного обеспечения вообще и о создании сайтов в частности. Приходится тратить гораздо больше времени на обслуживание и изменение вашего сайта, чем на его создание. По этой причине вам необходимо планировать поддержку и развитие вашего продукта при его разработке.

Если бы наш сайт, приспособленный для изменений, с самого начала был разработан с применением шаблонов проектирования, то не только бы он сохранил свою восприимчивость к переменам, но и нам не пришлось бы полностью отказываться от его текущего дизайна и начинать все с нуля ради его дальнейшего развития. То есть только часть

планирования должна быть обращена к изменениям при относительно стабильной структуре вашего сайта. Вам также потребуется план его глобального развития, которое будет превосходить все то, что вы изначально запланировали изменять. Поскольку количество материала для ваших последующих плановых изменений может еще больше увеличиться.

Планирование только для поддержки

Хотя наш сайт, описываемый как «созданный для изменений», и оправдал свое название, он не смог эволюционировать. Его легко было обслуживать, поскольку основной его функцией было загружать изображение, текстовый файл и позиции меню для заданного продукта. Изменение или добавление на него новых продуктов путем изменения названия кнопки или добавления новой кнопки на его страницу было также весьма несложным делом. Так что у него была небольшая расширяемость в том смысле, что он не был ограничен одним продуктом.

Однако к серьезным расширениям он был не готов. Например, добавление блога на сайт или изменение способа работы его меню потребовало бы такого же количества переделок, что и написание этого сайта с нуля. То есть изменить продукты на сайте было просто, а изменить сам сайт – нет. Структурно этот сайт выглядел так, как представлено на рис. 1.9.

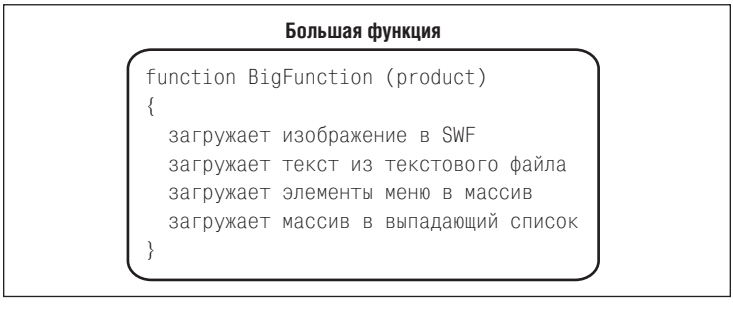


Рис. 1.9. *Большая функция*

В данном случае совершенно не важно, была ли большая функция написана непосредственно или реализована с помощью наследования, делегирования или композиции. У нее не было другой возможности, кроме параметра, которая говорила бы ей, что надо отобразить. Изменение ее алгоритма могло привести к полному хаосу в ее работе. Можно сказать, что она была негибкой, поскольку умела делать только одно и только одним способом. Кроме того, большая функция могла считаться хрупкой, поскольку вся работа сайта зависела от одной процедуры, которая была не приспособлена для общения с новыми элементами приложения. В любом случае она не способствовала гибкости всего сайта, и мы должны будем ее переделать.

Добавление расширяемости с помощью гранулярности

Наш план использования большой функции, несмотря на ее ограниченную гибкость, в конечном итоге обречен на провал. Чтобы избежать подобного тупика с неизменяемым приложением, вам необходимо заранее предусмотреть в вашем проекте его некоторую гранулярность. В данном контексте термин *гранулярность* означает ограничение количества функциональности, возлагаемой на ваши классы. Компромисс между функциональностью и гранулярностью состоит в том, что чем больше функциональности имеет некоторый класс, тем больше он может сделать самостоятельно. Не будем забывать, что большинство классов, которые мы создаем, добавляют еще больше функциональности к нескольким встроенным классам. Однако иногда чем меньше, тем лучше. Чем меньше функциональности имеет каждый класс, тем больше таких компонентов может использовать ваше приложение для воплощения своей итоговой функциональности. Рисунок 1.10 показывает, как может выглядеть подобная гранулярность.

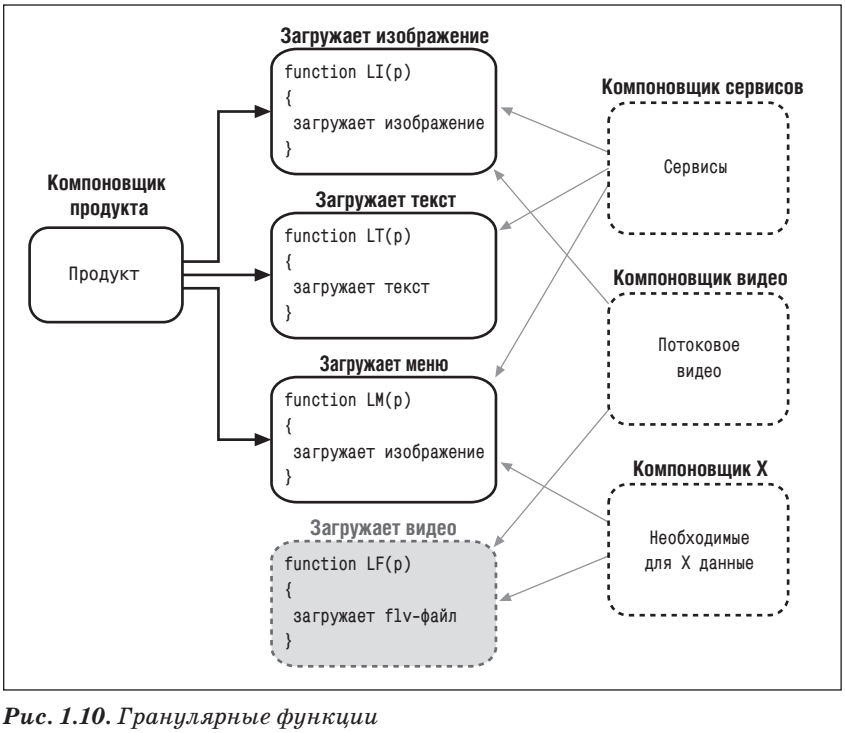


Рис. 1.10. *Гранулярные функции*

Наша большая функция из предыдущего раздела была разбита на три меньшие (более гранулярные) функции. С помощью композиции исходная функциональность большой функции была продублирована. Однако гранулярность дает программисту много новых возможностей на

будущее. В случае разработки для реального мира ваш проект должен «*заглядывать за горизонт*». То есть у вас должен быть план как для возможных, так и для неожиданных изменений.

Рисунок 1.10 показывает одно из возможных расширений нашего приложения (обозначенное пунктирными линиями). Преимущество гранулярности состоит в том, что новые классы могут использовать некоторые или все обособленные (гранулированные) функции. С одной большой функцией из предыдущего раздела это было бы невозможно. Кроме того, новые функции могут быть добавлены в проект и использоваться вместе с уже существующими.

План для вашего приложения: это не ваше дитя

Дженнифер Тидвел (Jennifer Tidwell) в своей лекции о дизайне интерфейсов напоминает разработчикам, что, как бы там ни было, но вовсе не они являются аудиторией (или заказчиками в данном случае) для своих разработок. Прежде чем доставать ваши любимые средства разработки, вам очень важно поговорить с заказчиком и выяснить, чего именно он хочет. Образ программиста, как парня со странной прической, сидящего в подвале (которому нельзя разрешать разговаривать с нашими клиентами), больше не является работоспособной моделью, особенно когда речь идет о проектировании веб-сайтов.

Поскольку всемирная паутина и используемые ею технологии постоянно изменяются, о чем свидетельствуют, например, перемены во Flash и ActionScript в последние годы, требования к программам, работающим в ней, также постоянно меняются. Поскольку разработчик лучше, чем кто-либо другой знает возможности и пределы изменений для своих приложений, он должен быть частью вышеуказанного процесса. В больших фирмах эта роль возлагается на дизайнеров графики, интерфейсов, человеко-машинного взаимодействия и управления информацией. В меньших фирмах разработчик может иметь несколько ролей, но, как бы там ни было организовано, разработчики должны участвовать в переговорах с заказчиком, чей бизнес зависит от исполнения проекта. Чем лучше разработчики будут понимать, чего хочет клиент, и чем лучше им удастся обсудить возможности и ограничения программ для достижения его целей, тем больше вероятность, что созданные приложения будут удовлетворять его требованиям.

Использование ООП и шаблонов проектирования для удовлетворения требований заказчика

Объектно-ориентированное программирование и шаблоны проектирования помогают разработчикам программ в создании сайтов, удовлетворяющих текущим и перспективным требованиям их заказчиков.

Процесс поддержки сайта в хорошем состоянии зависит как от возможностей по его регулярному обновлению, так и от возможностей по его дальнейшему расширению по мере необходимости. Слишком часто разработчики думают о своих сайтах, как о чем-то статичном, в то время как сайты являются динамичными, «живыми» сущностями, или, по крайней мере, так следует о них думать. Шаблоны проектирования содержат в себе средства (если угодно, архитектурные особенности), которые позволяют поддерживать «жизнь» сайтов.

Гибкость программного обеспечения является следствием возможности повторно использовать его компоненты. ООП и шаблоны проектирования разрабатывались с целью поддержки гибкости и повторного использования кода, поэтому, с практической точки зрения, их использование в веб-приложениях более чем оправданно. Так что шаблоны проектирования – это скорее не жесткие правила для создания крупных приложений, а гибкие средства по созданию именно таких сайтов, которые нужны вашим заказчикам.

Выбор нужного шаблона проектирования

Выбор наилучшего шаблона проектирования для каждой конкретной ситуации – это скорее искусство, чем некоторая жесткая формула. Вы увидите, что мы включили в данную книгу множество примеров такого выбора, и некоторые из них будут похожи друг на друга, но с разными шаблонами проектирования. Причина этого заключается в том, что большинство задач по разработке программ может быть решено несколькими методами. При этом один подход может показаться хорошим и естественным, а другой – более подходящим для данной ситуации. Рассмотрим пример. Пусть в нашем проекте требуется выбрать шаблон проектирования для проигрывателя видео, который должен будет его воспроизводить, записывать, останавливаться и делать паузу, используя Flash Media Server 2. По опыту другого похожего проекта решение по реализации такого проигрывателя должно лежать где-то в области автоматов с конечным числом состояний. Совпадение между тем, что должно быть сделано, и этой концепцией кажется просто идеальным. Отсюда остается всего один шаг до шаблона проектирования Состояние. Он был протестирован как решение такой задачи, прекрасно для нее подошел и имеет необходимый запас гибкости для дальнейших модификаций, поэтому он и был окончательно утвержден в качестве решения данной проблемы.

Как вы увидите, все шаблоны проектирования в данной книге разбиты на три части: порождающие, поведенческие и структурные. В каждой части книги описывается своя категория таких шаблонов проектирования. А главы внутри частей описывают, как создавать конкретные шаблоны на языке ActionScript 3.0, и предоставляют несколько подробно описанных примеров их применения.

«Банда четырех» разбила шаблоны проектирования на категории по цели их применения, а также проклассифицировала их по их *области применения*, описывающей, применяется ли данный шаблон к объекту или к классу. Когда мы выбирали шаблоны проектирования для данной книги, мы старались подобрать представителей из каждой ячейки матрицы их классификации. В табл. 1.1 представлены шаблоны, которые мы в итоге выбрали, в соответствии со своими целями и областями применения.

Таблица. 1.1. Классификация шаблонов проектирования

		Назначение		
		Порождающие	**Структурные**	**Поведенческие**
Область применения	Класс	Фабричный метод	Адаптер (класс)	Шаблонный метод
	Объект	Одиночка	Адаптер (объект) Компоновщик Декоратор	Команда Обозреватель Состояние Стратегия

Дальнейшее освоение ООП через шаблоны проектирования

Хотя вся эта глава была посвящена основным концепциям ООП и предназначалась тем, кто относительно мало знаком с этой темой, дальнейшее изучение шаблонов проектирования также должно помочь в освоении ООП. Мы осмелимся добавить, что если это ваше первое знакомство с ООП, то здесь вы найдете то, что некоторые считают *правильным* способом его изучения. (Кое-кто даже считает, что это *единственный* способ изучения ООП.) Мы потратили столько времени и сил на ООП, поскольку значительная часть из программистов на ActionScript могла еще не ознакомиться с ним в достаточной степени. В конце концов структуры ООП появились в языке ActionScript относительно недавно.

Но даже если вы уже «закаленный солдат» ООП, знающий его из языка ActionScript или какого-либо другого (например, Java или C++), мы надеемся, что наше обсуждение шаблонов проектирования поможет вам лучше использовать концепции ООП в практике применения этих шаблонов.

Каким бы ни был ваш исходный уровень знаний по ООП, «банда четырех» рекомендует изучение следующего списка шаблонов проектирования для тех из вас, у кого относительно мало опыта в объектно-ориентированном программировании:

- Адаптер (глава 5)
- Компоновщик (глава 6)

- Декоратор (глава 4)
- Фабричный метод (глава 2)
- Наблюдатель (глава 8)
- Стратегия (глава 11)
- Шаблонный метод (глава 9)

Если некоторые из этих шаблонов кажутся вам устрашающими, не переживайте. Мы не знаем никого, кто бы полностью освоил шаблоны проектирования с первого захода, включая шаблоны, рекомендованные «бандой четырех». Да и мы сами поняли не все сразу. Однако если вы будете использовать, изменять и экспериментировать с нашими примерами в процессе изучения приведенных в этих главах концепций ООП и шаблонов проектирования, то, мы убеждены, со временем вы будете так же, как мы, смотреть на шаблоны проектирования как на лучших друзей, которые только могут быть у программиста.

II

Порождающие шаблоны

Если бы Господь Всемогущий посоветовался со мной,
прежде чем приступать к созданию всего этого,
я бы порекомендовал ему сделать что-то попроще.

Приписывается Альфонсо Мудрому
(Король Кастилии в XIII веке и вообще мудрый человек)

Жить – значит меняться, меняться –
значит взрослеть, взрослеть – значит всякий раз
бесконечно творить себя заново.

Генри Бергсон

Ничто так не способствует
созданию будущего, как смелые мечты.

Виктор Гюго

В этой первой части рассматриваются шаблоны проектирования, которые абстрагируют процесс создания представителей классов. Порождающие шаблоны отделяют процесс использования объекта от его создания. Это разделение осуществляется с помощью инкапсуляции. Обычный класс использует эту инкапсулированную информацию, но напрямую ею не владеет. При этом обычный класс не видит, как создается представитель нужного ему класса.

Рисунок **II.1 показывает два основных свойства порождающих шаблонов.**

Два порождающих шаблона, которые мы будем здесь рассматривать, очень сильно различаются между собой. Шаблон Фабричный метод решает проблему жестких связей между клиентом и продуктом (которым

он пользуется) путем делегирования процесса создания отдельному методу, называемому *фабрика*. Отсюда и название этого шаблона – Фабричный метод. Шаблон Одиночка нацелен на создание *одного* и только одного представителя контролируемого им класса, что объясняет происхождение его названия. Шаблон проектирования Одиночка довольно часто используется вместе с другими шаблонами в качестве контролера, когда программисту необходимо, чтобы только один из элементов другого шаблона присутствовал в его программе.

Глава 2 «Шаблон Фабричный метод».

Глава 3 «Шаблон Одиночка».

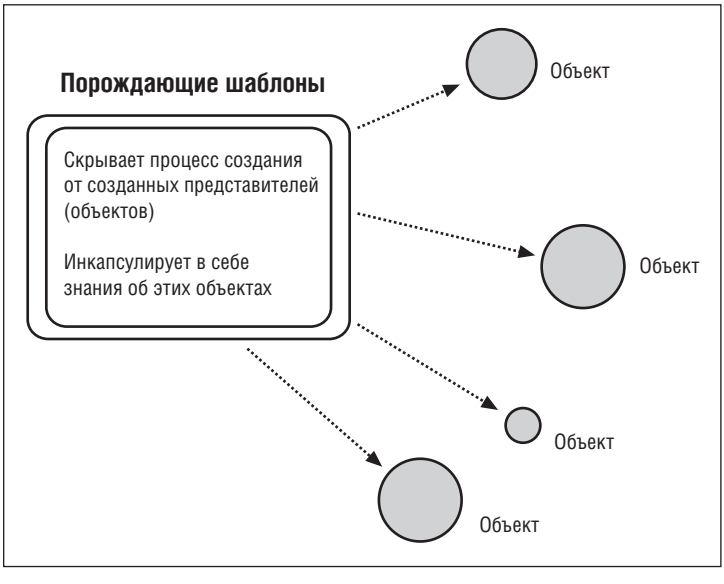


Рис. II.1. *Модель порождающих шаблонов*

2

Шаблон Фабричный метод

По мере того как усложняются эксперименты, возрастает потребность во взаимодействии с внешними производственными мощностями, и современная лаборатория стремится сильнее походить на фабрику и использовать все большее количество простых рабочих.

Джон Десмонд Бернал

Средневековые университеты смотрели в прошлое, они провозглашали себя хранилищами старых знаний. Современные университеты смотрят в будущее и являются фабриками новых знаний.

Томас Хаксли

Каждому, кто попадает в сборочный цех фабрики, кажется, что он оказался в аду.

Уистэн Хью Оден

Что такое шаблон Фабричный метод?

Одним из наиболее часто используемых слов в объектно-ориентированном программировании (ООП) является ключевое слово `new`, с помощью которого создаются объекты – представители конкретных классов. В приложении на ActionScript, имеющем множество различных классов, может многократно встречаться код, подобный нижеследующему:

```
public class Client
{
    public function doSomething()
    {
        var object:Object = new Product();
```

```
            object.manipulate();
        }
    }
```

Класс `Client` создает нового представителя класса `Product` и присваивает его переменной `object`. Это абсолютно нормально, однако вместе с тем создается несколько *связей* или *зависимостей* между классами `Client` и `Product`. Проще говоря, класс `Client` зависит в своей работе от класса `Product`. Любые изменения в классе `Product`, связанные с изменением его имени, или изменение числа передаваемых ему *параметров* потребуют изменений и в классе `Client`. Ситуация значительно усложняется, когда существует много таких клиентов, использующих класс `Product`, и приводит к необходимости внесения изменений в большое количество мест кода программы.

Решением этой общей проблемы является ослабление жестких связей между клиентом и конкретным классом, который он использует. Именно для такой ситуации шаблон Фабричный метод предлагает надежное решение. Он вводит посредника между клиентом и конкретным классом. Посредник называется классом-*создателем*. Он предоставляет клиенту объекты без указания конкретного класса для их создания. Это осуществляется с помощью делегирования операции создания объекта специальному методу создателя, называемому *фабрикой*. Основной задачей данного метода является создание представителя объекта и передача его адресату.

Модель шаблона Фабричный метод

На рисунке 2.1 показана высокоуровневая модель шаблона Фабричный метод. Многочисленные *клиенты* могут использовать метод-фабрику в *создателе* для получения доступа и использования нескольких *продуктов*. В данной модели ясно видна посредническая сущность создателя – он сводит создание множества типов продуктов к обращению в одну общую точку.

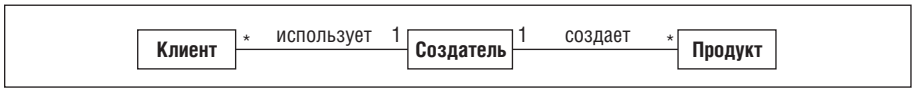


***Рис. 2.1.** Логическая модель шаблона Фабричный метод*

Данная модель не показывает, как клиент может использовать Фабричный метод для получения доступа к объектам без указания их конкретных классов. Давайте напишем некоторый код для создателя, который создает и возвращает объекты-продукты.

```
public class Creator
{
    public static function simpleFactory(product:String)
```

```
        {
            if (product == "p1")
            {
                return new product1();
            } else if (product == "p2") {
                return new product2();
            }
        }
    }
```

Параметризованный метод `simpleFactory()` («простая фабрика») создает представителей классов-продуктов и возвращает их обратно. При вызове этого метода клиент передает ему идентификаторы продуктов, которые в нашем случае являются значениями строки `String` – `p1` и `p2`. Все это ослабляет связь между клиентом и нужными для него классами-продуктами. Однако данный очень часто используемый фрагмент кода не дает такой гибкости и возможностей по повторному использованию кода, какие обеспечивает шаблон Фабричный метод. Подобный фрагмент кода обычно называют *простой фабрикой*.

Чтобы добавить еще один продукт, нам придется изменить код метода `simpleFactory()` и ввести в него еще одну ветку оператора `if`. Это противоречит принципу ООП «открытый-закрытый», который гласит, что код классов и методов должен быть *открыт для расширений*, но *закрыт для модификаций*. Одним из основных преимуществ шаблона Фабричный метод является как раз то, что он открыт для необходимых расширений. Давайте посмотрим на диаграмму классов полного шаблона Фабричный метод.

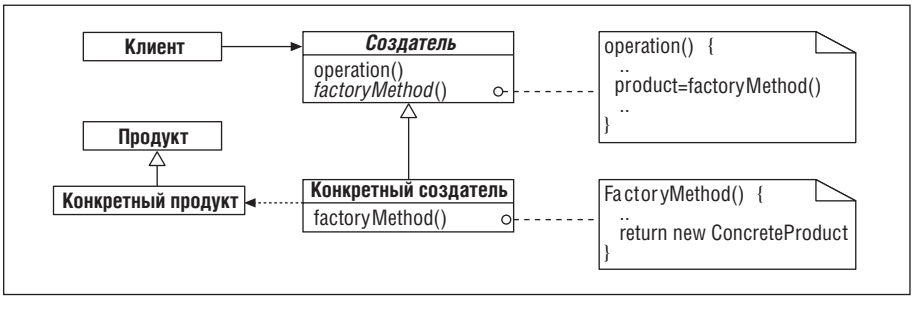


Рис. 2.2. *Диаграмма классов шаблона Фабричный метод*

И создатель, и продукт определяются *интерфейсами*, а не конкретными классами (интерфейсы на диаграмме обозначены наклонным шрифтом). Как известно, интерфейсы определяют для классов возвращаемый тип и сигнатуру методов. Классы, которые имплементируют интерфейс, обязаны реализовать методы, указанные в этом интерфейсе. Чистый интерфейс не содержит в себе никаких реализаций для объявленных методов. Однако существует особый вид интерфейсов – *аб-*

страктные интерфейсы. В них могут содержаться имплементации по умолчанию для некоторых методов. Их называют также *абстрактными классами*; они не могут создавать конкретных своих представителей, но могут расширяться своими подклассами.

Класс Creator на рис. 2.2 является абстрактным классом. В нем объявляется *фабричный метод* (называемый в данном случае factoryMethod()), в котором и происходит большая часть работы. Данный фабричный метод определен как *абстрактный метод* без какой-либо имплементации. Это заставляет подклассы Creator (классы, которые расширяют абстрактный класс, такие как ConcreteCreator) обязательно имплементировать данный фабричный метод. Класс Creator к тому же определяет и имплементирует публичный метод (именуемый в данном случае operation()), который вызывает фабричный метод.

```
operation()
{
    ..
    product = FactoryMethod()
    ..
}
```

Метод operation() вызывает FactoryMethod() для создания объекта – продукта операции.

```
factoryMethod() {
    ..
    return new ConcreteProduct
}
```

Основной обязанностью фабричного метода является создание конечного объекта и передача его адресату. Интересно здесь то, что хотя factoryMethod() и объявлен в абстрактном классе Creator, имплементируется он только в классе ConcreteCreator. Следовательно, именно класс ConcreteCreator знает о конкретных классах конечного продукта, но полностью скрывает их от клиента.

Причина использования интерфейсов и абстрактных классов станет вам ясна после просмотра реального приложения, в котором допускается добавление нового продукта и его конкретного создателя путем расширения, а не замены существующего кода. Это важный момент в идеологии ООП, поскольку позволяет добавлять в программу новую функциональность без опасения что-либо нарушить. Класс ConcreteCreator расширяет собой абстрактный класс Creator. Интерфейс Product может быть либо чистым интерфейсом, либо абстрактным классом в зависимости от того, нужна ли в нем некоторая функциональность по умолчанию, которая должна быть реализована для всех продуктов.

Клиент получает доступ к классам ConcreteProduct через интерфейс Creator. Чтобы вынудить клиентов обращаться за продуктами через фа-

бричные методы, сами классы продуктов стараются скрыть от внешнего доступа к ним. Мы можем посмотреть, как это все реализуется в ActionScript на следующем далее минималистском примере.

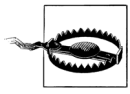
Абстрактные классы в ActionScript 3.0

Прежде чем писать пример реализации шаблона Фабричный метод, нам нужно хорошенько разобраться с темой *абстрактных классов*, а более конкретно – с отсутствием их поддержки в ActionScript 3.0. В шаблоне Фабричный метод все создатели определяются через абстрактные классы, и другого способа для этого не существует. На самом деле большая часть полезных свойств данного шаблона может быть объяснена такой абстрактностью.

По определению абстрактные классы не могут создавать своих представителей. Они должны только расширяться своими подклассами. Они могут содержать в себе абстрактные методы или объявления нереализованных методов, которые их подклассы должны имплементировать. Методы, которые имплементируются в абстрактных классах, в большинстве случаев будут являться их поведением по умолчанию, а большая часть класса останется неимплементированной. До того как класс, произошедший от абстрактного класса, сможет создать своего представителя, он должен реализовать все неимплементированные методы. Преимуществом использования абстрактного класса является то, что дочернему классу не нужно имплементировать метод, если имплементация по умолчанию в абстрактном классе вполне подходит.

Использование в качестве класса `Creator` абстрактного класса позволяет нам сосредоточиться только на тех его элементах, которые должны измениться, и оставить остальные функционирующими по умолчанию. Это снижает сложность реализации (одно из ключевых преимуществ ООП). Например, в нашей модели мы можем использовать имплементацию метода `operation()` по умолчанию, но заместить *`FactoryMethod`*`()`.

К сожалению, ActionScript 3.0 не поддерживает явным образом абстрактные классы в своем синтаксисе. Их можно реализовывать как обычные классы ActionScript без проверки на создание представителей и обязательную имплементацию абстрактных методов. В случае языков, в которых есть абстрактные классы, такие проверки проводят на этапе компиляции. Мы можем добавить в наши обычные классы ActionScript 3.0 код, который будет осуществлять данные проверки во время исполнения и сообщать при необходимости об ошибке. Очень важно помечать классы и методы, которые должны вести себя как абстрактные, специальными комментариями. Должно быть подчеркнуто, что именно на программиста, а не на компилятор возложена ответственность проверить, что все методы, которые должны вести себя как абстрактные, имплементированы.



В дальнейшем мы будем определять *абстрактные* классы через *обычные* классы, полагая, что их представители не будут создаваться в программах, и они будут только расширяться своими подклассами. *Абстрактные методы* будут определяться как функции, которые будут выдавать ошибку `IllegalOperationError` при обращении к их незамещенным версиям. И абстрактные классы, и абстрактные методы в них будут помечаться соответствующими комментариями.

Минималистский пример

Мы покажем здесь минималистский пример применения шаблона фабричного метода на ActionScript 3.0 с использованием Flash CS3. Окно проекта для данного примера, изображенное на рис 2.3, отражает его структуру файлов.

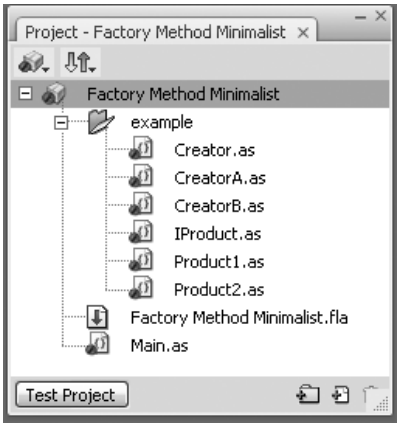


***Рис. 2.3.** Окно проекта для минималистского примера*

Factory Method Minimalist.fla – это документ Flash, чьим классом в данном проекте является `Main`. *Класс документа* – это новая возможность во Flash CS3, которая может быть установлена с помощью панели свойств документа Flash (файла типа *.fla*). Он представляет класс, чей конструктор будет автоматически запущен на исполнение проигрывателем Flash сразу после загрузки *swf*-файла. Класс `Main` описывается в файле ActionScript с именем *Main.as*. Наш проект содержит также *пакет* (package), называемый `example`. Пакеты позволяют вам собрать ваши классы вместе для упорядочивания вашего кода. Они также дают возможность контролировать видимость классов и их методов вне пакетов, указывая идентификаторы последних. Это минимизирует конфликты именования, которые могут возникать при разработке сложных приложений. В данном примере мы будем использовать пакеты,

чтобы скрыть классы продуктов от прямого доступа к ним со стороны клиентов.

Пакет `example` содержит в себе классы `Product1` и `Product2`, которые имплементируют интерфейс `IProduct`. Он также содержит *абстрактный* класс `Creator`, который расширяется двумя подклассами `CreatorA` и `CreatorB`. Каждый класс описывается в своем собственном файле, как это положено в ActionScript 3.0. Давайте для начала рассмотрим классы продуктов.

Классы продуктов

Примеры 2.1 – 2.3 показывают интерфейс продуктов и их конкретные классы. Все три класса принадлежат к пакету `example`, что указано в их описаниях после ключевого слова `package`. Интерфейс `IProduct` определяет в себе обязательный метод с именем `manipulate()`, который потом имплементируется в обоих конкретных классах продуктов. Ввиду минималистичности нашего примера классы продуктов реально практически ничего не делают. Однако обратите внимание на их атрибуты.

Пример 2.1. IProduct.as

```
package example
{
    public interface IProduct
    {
        function manipulate():void;
    }
}
```

Пример 2.2. Product1.as

```
package example
{
    internal class Product1 implements IProduct
    {
        public function manipulate():void
        {
            trace("Doing stuff with Product1");
        }
    }
}
```

Пример 2.3. Product2.as

```
package example
{
    internal class Product2 implements IProduct
    {
        public function manipulate():void
        {
```

```
            trace("Doing stuff with Product2");
        }
    }
}
```

Классы продуктов определены с атрибутом *internal* (являющимся атрибутом класса по умолчанию для ActionScript 3.0). Это означает, что они не видны вне своего пакета `example` и к ним можно обращаться только внутри него. Теперь рассмотрим классы создателей.

Классы создателей

В примерах с 2.4 по 2.6 показаны классы создателей. Как и классы продуктов, все классы создателей принадлежат к пакету `example`, но при этом они определены как *публичные* классы, что позволяет осуществлять доступ к ним извне их пакета. Каждый класс создателя, о чем свидетельствует комментарий к нему, должен вести себя как абстрактный класс. Он должен только расширяться своими подклассами и не создавать своих представителей. В нем также определяется фабричный метод, который должен вести себя как абстрактный метод. Обратите внимание, два обычных класса `CreatorA` и `CreatorB` расширяют собой класс `Creator`. Они также замещают и имплементируют его фабричный метод `factoryMethod()`, который в них возвращает соответствующий объект-продукт. Вдобавок `factoryMethod()`, изначально определенный в классе `Creator`, возвращает значение `null`, чтобы избежать ошибки во время компиляции. Он также вызывает ошибку `IllegalOperationError` при своем непосредственном вызове. Таким образом осуществляется контрольная проверка замещения данного метода во время исполнения программы.

Пример 2.4. Creator.as

```
package example
{
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class Creator
    {
        public function doStuff():void
        {
            var product:IProduct = this.factoryMethod();
            product.manipulate();
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        protected function factoryMethod():IProduct
        {
```

```
            throw new IllegalOperationError("Abstract method:
                                        must be overridden in a subclass");
            return null;
        }
    }
}
```

Пример 2.5. CreatorA.as

```
package example
{
    public class CreatorA extends Creator
    {
        override protected function factoryMethod():IProduct
        {
            trace("Creating product 1");
            return new Product1(); // возвращает конкретный продукт
        }
    }
}
```

Пример 2.6. CreatorB.as

```
package example
{
    public class CreatorB extends Creator
    {
        override protected function factoryMethod():IProduct
        {
            trace("Creating product 2");
            return new Product2(); // возвращает конкретный продукт
        }
    }
}
```

Заметьте, что метод `doStuff()` в классе `Creator` определен как *публичный*, поскольку нам нужно разрешить доступ к нему клиентов извне пакета. А метод `factoryMethod()` определен как *защищенный* (`protected`). Такой атрибут делает этот метод видимым только для данного класса или его наследников. Наконец давайте рассмотрим класс документа, называемый `Main`, который является в диаграмме классов на рис. 2.2 клиентом, использующим создателя для получения продуктов.

Клиенты

Класс документа `Main` не принадлежит к именованному пакету. Следовательно, он должен импортировать пакеты, содержащие классы, которые он использует. Пакет `example` содержит класс `Creator`, поэтому он должен быть импортирован. Класс `Main` является клиентом для шаблона Фабричный метод, который был описан в его модели высокого уровня, представленной на рис 2.1.

Пример 2.7. Main.as

```
package
{
    import flash.display.Sprite;
    import example.*;

    /**
    *    Главный класс
    *    @ назначение: класс документа
    */
    public class Main extends Sprite
    {
        public function Main()
        {
            // реализация конкретных создателей
            var cA:Creator = new CreatorA();
            var cB:Creator = new CreatorB();

            // создатели работают с разными продуктами,
            // хотя и выполняют одну и ту же операцию
            cA.doStuff();
            cB.doStuff();
        }
    }
}
```

После запуска этой программы мы получим на ее выходе следующее:

```
Creating product 1
Doing stuff with Product1
Creating product 2
Doing stuff with Product2
```

Давайте посмотрим на наш код еще раз и проследим по нему, как мы получили такой результат. Наш клиент (`Main` класс документа) ничего не знает о классах продуктов. Он имеет представление только о классах создателей и о том, что они делают. Следовательно, клиент создает представителей создателей `CreatorA` и `CreatorB` и просит их выполнить метод `doStuff()`. Класс `Creator`, который ведет себя как абстрактный, знает, как сделать `doStuff()`, но он должен передать своим подклассам определение того, какой продукт при этом будет использоваться. Метод `doStuff()` вызывает фабричный метод `factoryMethod()`, возвращающий объект-продукт. Потом вызывается метод `manipulate()` полученного объекта-продукта.

Основной задачей подклассов `CreatorA` и `CreatorB` является создание и возвращение объектов-продуктов. Они знают, с какими классами продуктов им следует работать, и замещают в себе фабричный метод, чтобы создать в нем представителя нужного класса-продукта и вернуть его в метод `doStuff()`. Все знания о создании конкретного объекта были инкапсулированы внутри классов создателей.

Таким образом, шаблон Фабричный метод создает что-то вроде разделительной стены между клиентом и конкретным классом продукта, которым он пользуется. Является ли эта стена абсолютно непроницаемой? Полностью ли мы выполнили то, что хотели? Действительно ли классы продуктов доступны только через создателей и никак иначе? Давайте это проверим.

Сокрытие классов продуктов

Давайте проверим, действительно ли классы продуктов скрыты и доступны только через классы создателей. Мы можем попытаться создать представителя продукта путем непосредственного обращения к классу создателя и его фабричному методу. Добавьте следующие команды в класс `Main` файла *Main.as* (см. пример 2.7), чтобы попробовать напрямую создать продукты:

```
// реализация конкретных продуктов
var p1 = new Product1();
var p2 = CreatorB.factoryMethod();[1]
```

При попытке запуска нашего проекта возникнут следующие ошибки этапа компиляции.

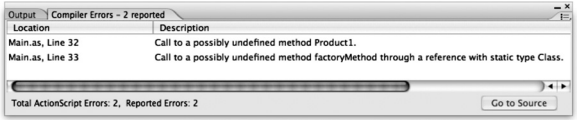


***Рис. 2.4.** Ошибки компиляции при попытке прямого доступа к классам продуктов* [2]

Эти ошибки показывают нам, что классы продуктов и фабричный метод для клиента не видны. Классы продуктов инкапсулированы внутри своего пакета, что было определено их атрибутом (по умолчанию) `internal` («внутренний»). Внутренние классы доступны только внутри своего пакета. Аналогичным образом фабричный метод `factoryMethod()` не является доступным, поскольку он был описан как `protected` («защищенный»). Следовательно, доступ к классам продуктов возможен только через *публичный* метод `doStuff()`.

---

[1] Эта строка будет генерировать ошибку вне зависимости от того, закрыты внутренние классы шаблона или нет, поскольку в данном примере вызов метода `factoryMethod` производится как вызов статического метода класса `CreatorB`, а на самом деле `factoryMethod` статическим не является. – *Прим. науч. ред.*

[2] Main.as, строка 32: Вызов возможно неопределенного метода Product1 Main.as, строка 33: Вызов возможно неопределенного метода factoryMethod посредством ссылки со статическим типом Class. – *Прим. науч. ред.*

Вы можете задаться вопросом, зачем вообще использовать такие сложные схемы только ради того, чтобы предотвратить прямой доступ клиента к созданию представителей некоторых классов. Ведь сделать наоборот гораздо проще. Клиент мог бы создать `Product1` и `Product2` и сам вызвать у них метод `doStuff()`. Давайте обсудим эту тему после рассмотрения еще одного простого приложения, использующего шаблон Фабричный метод. Преимущества нашего шаблона будет гораздо проще увидеть в более реальном контексте, чем при использовании абстрактных продуктов и их создателей.

Пример: печатная мастерская

Давайте разработаем пример приложения, которое будет распределять печатные работы в некоторой вымышленной печатной мастерской с несколькими принтерами. Предположим, что клиенты приносят в такую мастерскую на переносных цифровых носителях то, что они хотят напечатать. Служащий принимает заказ, с помощью компьютера отправляет его на исполнение и выставляет счет в зависимости от типа печатной работы. Мы построим такое приложение во Flash и ActionScript 3.0 в качестве примера использования шаблона Фабричный метод.

Наша печатная мастерская – это небольшое предприятие только с двумя принтерами. У нас есть групповой производительный принтер и маломощный струйный принтер. Чтобы оптимизировать выполнение всех операций, менеджер нашей мастерской предлагает создать в ней раздельные центры печати. Один центр печати будет обрабатывать все объемные задания, которые будут отправляться на групповой принтер. А другой центр – малообъемные задания, которые будут распределяться на струйный принтер. Мы будем разрабатывать наше приложение с помощью шаблона Фабричный метод, поскольку мы слышали, что он позволяет обеспечить его гибкость и расширяемость в будущем, а наша

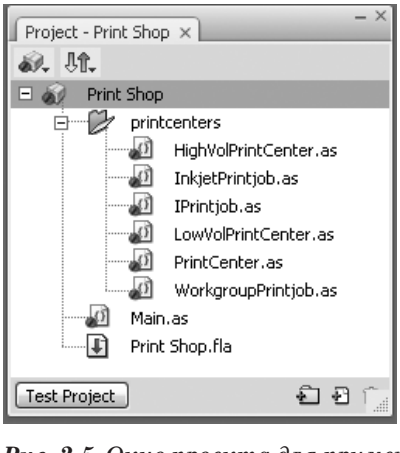


***Рис. 2.5.** Окно проекта для примера печатной мастерской*

мастерская надеется получить прибыль и приобрести в будущем еще несколько принтеров. Рисунок 2.5 показывает структуру классов для нашего примера приложения для такой печатной мастерской.

Наша структура классов очень похожа на аналогичную структуру для нашего минималистского примера, показанную на рис. 2.3. Продуктами в данном случае будут задания на печать, а их создателями – печатные центры (см. диаграмму на рис. 2.6).

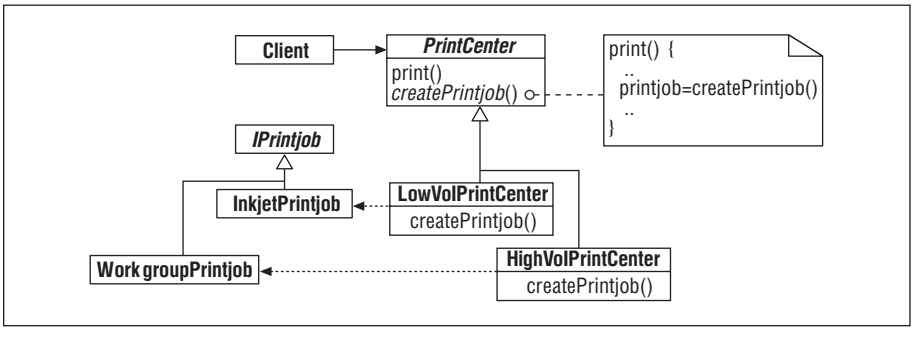


Рис. 2.6. *Диаграмма классов для примера печатной мастерской*

Классы продуктов: задания на печать

Хотя сейчас классы продуктов ничего не делают, в реальном приложении они будут инициировать выставление счета за работу и отправлять ее на сервер печати. Наши классы заданий на печать принадлежат к пакету `printcenters`. Пример 2.8 показывает интерфейс `IPrintjob`, определяющий в себе параметризованный метод с именем `start()`, который принимает на своем входе имя документа, отправляемого на печать.

Пример 2.8. IPrintjob.as

```
package printcenters
{
    public interface IPrintjob
    {
        function start(fn:String):void;
    }
}
```

Примеры 2.9 и 2.10 показывают два конкретных класса заданий на печать, которые имплементируют собой интерфейс `IPrintJobs`.

Пример 2.9. InkjetPrintjob.as

```
package printcenters
{
    internal class InkjetPrintjob implements IPrintjob
    {
```

```
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to inkjet printer");
        }
    }
}
```

Пример 2.10. WorkgroupPrintjob.as

```
package printcenters
{
    internal class WorkgroupPrintjob implements IPrintjob
    {
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to workgroup printer");
        }
    }
}
```

Классы создателей: центры печати

В примере 2.11 показан класс `PrintCenter`, который должен рассматриваться как абстрактный класс. Его фабричный метод называется `createPrintjob()`, а клиенты получают доступ к центрам печати через его метод `print()`.

Пример 2.11. PrintCenter.as

```
package printcenters
{
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class PrintCenter
    {
        public function print(fn:String):void
        {
            var printjob:IPrintjob = this.createPrintjob();
            printjob.start(fn);
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        protected function createPrintjob():IPrintjob
        {
            throw new IllegalOperationError("Abstract method:
                                        must be overridden in a subclass");
            return null;
        }
    }
}
```

Примеры 2.12 и 2.13 демонстрируют классы LowVolPrintCenter и LowVolPrintCenter, которые расширяют класс PrintCenter, а также замещают и имплементируют его фабричный метод createPrintjob().

Пример 2.12. LowVolPrintCenter.as

```
package printcenters
{
    public class LowVolPrintCenter extends PrintCenter
    {
        override protected function createPrintjob():IPrintjob
        {
            trace("Creating new printjob for the inkjet printer");
            return new InkjetPrintjob();
        }
    }
}
```

Пример 2.13. HighVolPrintCenter.as

```
package printcenters
{
    public class HighVolPrintCenter extends PrintCenter
    {
        override protected function createPrintjob():IPrintjob
        {
            trace("Creating new printjob for the workgropup printer");
            return new WorkgroupPrintjob();
        }
    }
}
```

Клиенты

Заметьте, клиенты не знают, какие конкретно принтеры есть в наличии в мастерской печати. Они только знают тип задания на печать, определяемый классом создателя. Основываясь на объеме принесенного заказчиком в мастерскую задания, приложение должно создать представителя соответствующего центра печати (LowVolPrintCenter или HighVolPrintCenter) и вызвать его метод print(), передав ему имя файла, в котором находится документ для печати.

Чтобы протестировать наш проект, мы запустим следующий код на стороне клиента.

```
var pcHighVol:PrintCenter = new HighVolPrintCenter();
var pcLowhVol:PrintCenter = new LowVolPrintCenter();

pcHighVol.print("LongThesis.doc");
pcLowhVol.print("ShortVita.doc");
```

Как и в нашем минималистском примере, мы получим следующий результат:

```
Creating new printjob for the workgropup printer
Printing 'LongThesis.doc' to workgroup printer
Creating new printjob for the inkjet printer
Printing 'ShortVita.doc' to inkjet printer
```

Отсюда ясно видно, что метод `print()` работает у разных объектов (а именно объектов `WorkgroupPrintjob` и `InkjetPrintjob`). Получилось элегантное решение, в котором клиент просто выбирает подходящий ему центр для печати и запрашивает распечатку задания. Центры печати берут на себя всю заботу о корректной печати этого задания. Очевидно также разделение между созданием объекта и его использованием. Создание объекта происходит с помощью фабричного метода `factoryMethod()`, а созданные им объекты используются с помощью вызова их метода `print()`. Создание объекта при этом полностью скрыто от клиента.

Расширение печатной мастерской

Реальную пользу от применения шаблона Фабричный метод можно увидеть при расширении или добавлении новой функциональности к существующему приложению. Поскольку дела в нашей печатной мастерской идут весьма хорошо, ее управляющий решает добавить в нее еще один замечательный многофункциональный принтер, имеющий сортировочный лоток, встроенный степлер и возможность двусторонней печати. Как нам добавить этот новый принтер в наше существующее приложение? Нам потребуется новый центр печати в нашей модели печатной мастерской для всех тех клиентов, кому нужны дополнительные опции, такие как скрепление документов или двусторонняя печать. В терминах программы нам придется создать новый подкласс класса `PrintCenter`, чтобы обрабатывать такие задания на печать. Мы назовем его `FancyPrintCenter` (см. пример 2.15). Еще нам придется создать подкласс `PrintJob` и разработать на его основе новый конкретный класс, называемый `MultifunctionPrintJob` (см. пример 2.14), который будет передавать задания на новый принтер. Давайте посмотрим, какие изменения нам нужно сделать для всего этого.

Пример 2.14. MultifunctionPrintJob.as

```
package printcenters {

    internal class MultifunctionPrintJob implements IPrintjob {

        public function start(fn:String):void {
            trace("Printing '" + fn + "' to multifunction printer");
        }
    }
}
```

Пример 2.15. FancyPrintCenter.as

```
package printcenters {

    import printcenters.*;

    // Многофункциональный принтер (подкласс PrintCenter)
    public class FancyPrintCenter extends PrintCenter {

        override protected function createPrintjob():IPrintjob {
            trace("Creating new printjob for the multifunction printer");
            return new MultifunctionPrintJob();
        }
    }
}
```

Мы добавили в программу два новых класса, но больше ничего с кодом не делали. Мы добавили новый класс продукта путем имплементации интерфейса `PrintJob` и новый класс создателя расширением абстрактного класса `PrintCenter`. В этом и состоит большое преимущество шаблона Фабричный метод перед простой фабрикой, которая обсуждалась ранее. Поскольку мы объявили и класс продуктов, и класс их создателей через интерфейсы, у нас появилась возможность добавлять в программу новую функциональность без изменения существующего кода. Чтобы получить доступ к новому принтеру, клиенту всего лишь нужно создать представителя класса `FancyPrintCenter` и вызвать его метод `print()`.

Параметризованные фабричные методы

Все примеры, которые мы рассмотрели, использовали непараметризованные фабричные методы. *Непараметризованный* фабричный метод не требует каких-либо параметров в своем описании. *Параметризованные* фабричные методы требуют параметр, определяющий, какой именно вид продукта будет создан. Например, в нашем приложении для печатной мастерской можно передавать дополнительный *параметр* для фабричного метода, указывающий особенности данного задания на распечатку. Параметризованные фабричные методы позволяют более полно использовать инкапсуляцию и иллюстрируют полезность использования шаблона Фабричный метод. Мы расширим наш пример для печатной мастерской, чтобы включить в него параметризованный фабричный метод.

Расширенный пример: цветная печать

Дела в печатной мастерской идут просто замечательно! И ее управляющий решил добавить еще одну возможность, пользующуюся большим спросом, – цветную печать. Для этого были закуплены два принтера: цветной струйный принтер для малых объемов печати и цветной лазерный принтер для больших объемов. К сожалению, мы совершенно

не думали про цветную печать, когда планировали наше приложение. Как же нам в него добавить такую возможность? Хорошо, что мы теперь знаем о параметризованных фабричных методах.

У нас по-прежнему имеются два центра печати для больших и малых объемов. Однако теперь нам нужно указывать, какого вида будет эта печать. Нашим заказчикам может потребоваться цветная или черно-белая печать большого или малого объема. Чтобы обозначить вид конкретного задания на печать, мы можем передавать параметр в метод `print()`.

Новые классы продуктов

Мы добавили два новых класса продуктов для новых принтеров, называемые `ColorInkjetPrintjob` (пример 2.16) и `ColorLaserPrintjob` (пример 2.17). Повторим еще раз, что мы не меняем существующий код, а только расширяем наше существующее приложение, имплементируя интерфейс `IPrintJob`, описанный в примере 2.8.

Пример 2.16. ColorInkjetPrintjob.as

```
package printcenters
{
    internal class ColorInkjetPrintjob implements IPrintjob
    {
    public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to color inkjet printer");
        }
    }
}
```

Пример 2.17. ColorLaserPrintjob.as

```
package printcenters
{
    internal class ColorLaserPrintjob implements IPrintjob
    {
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to color laser printer");
        }
    }
}
```

Новые классы создателей: интеграция параметризованного Фабричного метода

Как и в случае с новыми классами продуктов, мы можем добавить параметризованный Фабричный метод в наше приложение без изменения существующего кода. Мы не будем изменять первоначальный аб-

страктный класс PrintCenter и все его производные классы LowVolPrintCenter и HighVolPrintCenter. Тогда мы не разрушим функциональность старого интерфейса, и использующие его клиенты будут продолжать спокойно работать. Мы определим новый абстрактный интерфейс, называемый NewPrintCenter, с параметризованным Фабричным методом (пример 2.18).

Пример 2.18. NewPrintCenter.as

```
package printcenters
{
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class NewPrintCenter
    {

        public function print(fn:String, cKind:uint):void
        {
            var  printjob:IPrintjob = this.createPrintjob(cKind);
            printjob.start(fn);
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        protected function createPrintjob(cKind:uint):IPrintjob
        {
            throw new IllegalOperationError("Abstract method:
                                            must be overridden in a subclass");
            return null;
        }
    }
}
```

Новый фабричный метод createPrintJob() принимает параметр типа uint, который означает вид требуемой печати (цветная или черно-белая). Метод print() также принимает этот дополнительный параметр.

Теперь нам нужно создать два обычных класса (примеры 2.19 и 2.20), которые будут расширять абстрактный класс NewPrintCenter для случаев больших и малых объемов печати.

Пример 2.19. NewLowVolPrintCenter.as

```
package printcenters
{
    public class NewLowVolPrintCenter extends NewPrintCenter
    {
        public static const BW        :uint =  0;
        public static const COLOR     :uint =  1;

        override protected function createPrintjob(cKind:uint):IPrintjob
```

```
        {
            if (cKind == BW)
            {
                trace("Creating new printjob for the b/w inkjet printer");
                return new InkjetPrintjob();
            } else if (cKind == COLOR) {
                trace("Creating new printjob for the color inkjet printer");
                return new ColorInkjetPrintjob();
            } else {
                throw new Error("Invalid low volume print job");
                return null;
            }
        }
    }
}
```

Пример 2.20. NewHighVolPrintCenter.as

```
package printcenters
{
    public class NewHighVolPrintCenter extends NewPrintCenter
    {
        public static const BW        :uint =  0;
        public static const COLOR     :uint =  1;

        override protected function createPrintjob(cKind:uint):IPrintjob
        {
            if (cKind == BW)
            {
                trace("Creating new printjob for the b/w workgroup printer");
                return new WorkgroupPrintjob();
            } else if (cKind == COLOR) {
                trace("Creating new printjob for the color laser printer");
                return new ColorLaserPrintjob();
            } else {
                throw new Error("Invalid high volume print job");
                return null;
            }
        }
    }
}
```

Мы создали новую иерархию связанных классов для новых центров печати с параметризованным фабричным методом. Новым абстрактным классом стал `NewPrintCenter`, который расширяется своими подклассами `NewLowVolPrintCenter` и `NewHighVolPrintCenter`. Обратите внимание на статичные публичные константы, обозначающие различные виды заданий на печать. Эти константы доступны для всех и должны использоваться как параметры, передаваемые методу `print()`.

Клиенты

Клиенты должны создать представителей новых центров печати, чтобы получить доступ к цветным принтерам. Теперь их метод `print()` принимает новый параметр, определяющий, цветная будет печать или черно-белая. В зависимости от этого параметра происходит работа с разными объектами по распечатке заказов.

```
var pcNewHighVol:NewPrintCenter = new NewHighVolPrintCenter();
var pcNewLowhVol:NewPrintCenter = new NewLowVolPrintCenter();

pcNewHighVol.print("LongThesis.doc", NewHighVolPrintCenter.BW);
pcNewHighVol.print("SalesReport.pdf", NewHighVolPrintCenter.COLOR);
pcNewLowhVol.print("ShortVita.doc", NewLowVolPrintCenter.BW);
pcNewLowhVol.print("SalesChart.xlc", NewLowVolPrintCenter.COLOR);
```

Старые классы центров печати `LowVolPrintCenter` и `HighVolPrintCenter`, которые реализовывали непараметризованный фабричный метод, продолжат свою работу. Они по-прежнему будут использовать классы заданий на распечатку `WorkgroupPrintjob` и `InkjetPrintjob` без каких-либо изменений. Таким образом, способность шаблона проектирования Фабричный метод поддерживать необходимые изменения в программе проявилась в данном примере самым очевидным образом.

Параллельные иерархии классов

Взгляните на диаграмму классов для нашего расширенного приложения для печатной мастерской (рис. 2.7). Вы можете там увидеть две различные иерархии классов продуктов. Классы `WorkgroupPrintjob` и `ColorLaserPrintjob` представляют задания на печать большого объема. Классы `InkjetPrintjob` и `ColorInkjetPrintjob` представляют распечатки малого объема. Вся информация об иерархиях этих классов инкапсулирована в их соответствующих классах создателей. Заметьте, что у клиентов нет прямого доступа к классам продуктов; последние могут быть доступны только

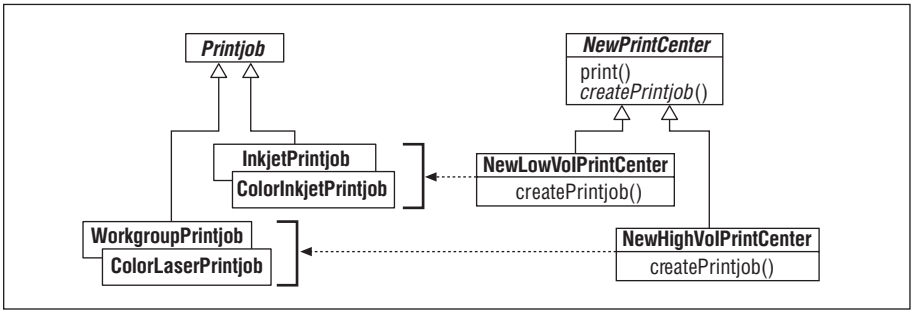


Рис. 2.7. *Диаграмма классов для расширенного примера печатной мастерской*

через классы создателей. Только `NewLowVolPrintCenter` и `NewHighVolPrintCenter` знают о своих иерархиях классов продуктов. Таким образом, мы можем считать шаблон Фабричный метод набором *параллельных иерархий классов*: конкретных классов создателей и их соответствующих продуктов, являющимся мощным средством для инкапсулирования знаний о структуре и управления зависимостями в программных приложениях.

Ключевые концепции ООП, использованные в шаблоне Фабричный метод

Внесение в приложение изменений, не предусмотренных при его проектировании, может иногда потребовать изменения существующего кода. Однако следует всеми силами избегать такой модификации работающего кода, поскольку она может привести к нежелательным последствиям, таким как введение в программу новых ошибок. Даже небольшие изменения в зависимом модуле могут повлечь за собой нарушение программы в нескольких местах, если имеет место жесткая связь между фрагментами кода.

Тем не менее при помощи хорошего объектно-ориентированного проектирования программ возможно создать код, который будет довольно хорошо переносить изменения. Шаблон Фабричный метод является замечательным решением этой традиционной проблемы. Он позволяет ослабить одни из наиболее часто встречающихся жестких связей, которые возникают, когда один класс создает представителя другого класса и использует получившийся объект. Конечно, создание представителей классов неизбежно, и написание кода без этого совершенно немыслимо. Так о чем же мы тогда здесь говорим?

Мы не планируем совсем устранять связь, возникающую при создании представителя конкретного класса. Мы можем управлять зависимостями между классами и ослаблять их связь. Для достижения данной цели шаблон Фабричный метод позволяет вам отделить процесс *создания* объекта от процесса его *использования*.

В этом состоит основная концепция данного шаблона проектирования. Клиент может *использовать* объект, созданный другим классом, но Фабричный метод управляет *созданием* этого объекта путем введения посредника, называемого классом-создателем, между клиентом и конкретным классом, представителя которого он хочет создать. Клиенту при этом не нужно знать или указывать имя класса, объект которого он хочет использовать, поскольку класс создателя инкапсулирует в себе эту информацию. Такая инкапсуляция позволяет управлять изменениями через расширение, как это было показано в примере с печатной мастерской.

В конечном счете, шаблон Фабричный метод позволяет разработчикам создавать слабо связанные структуры классов, которые выдерживают испытание изменчивыми требованиями к их приложениям.

Пример: фабрика спрайтов

ActionScript 3.0 содержит в себе класс `Sprite`, который является исходным строительным блоком для создания интерактивных объектов на сцене. Класс `MovieClip` теперь также происходит от класса `Sprite`. Шаблон Фабричный метод может оказаться очень полезным при разработке приложений, использующих спрайты во Flash. В таких приложениях очень часто добавляют новые спрайты, и их внешний вид и поведение изменяются на протяжении всего цикла разработки. Таким образом, управление зависимостями между спрайтами и всем остальным приложением может оказаться весьма кстати. Мы рассмотрим здесь простой пример приложения под названием `Shapes` («Фигуры»), которое будет управлять созданием спрайтов с помощью шаблона Фабричный метод, что позволит клиентам создавать спрайты без указания имени их класса.

Наше приложение, показанное на рис. 2.8, будет просто рисовать на Flash-сцене четыре различные фигуры, основанные на классе `Sprite`. Первое множество фигур будет состоять из *незаполненных* прямоугольников и кругов. Второе множество – из *заполненных* прямоугольников и кругов. Хотя такой пример и не является реалистичным приложением, он послужит нам основой для *вертикальной стрелковой игры*, которую мы создадим позднее в этой главе.

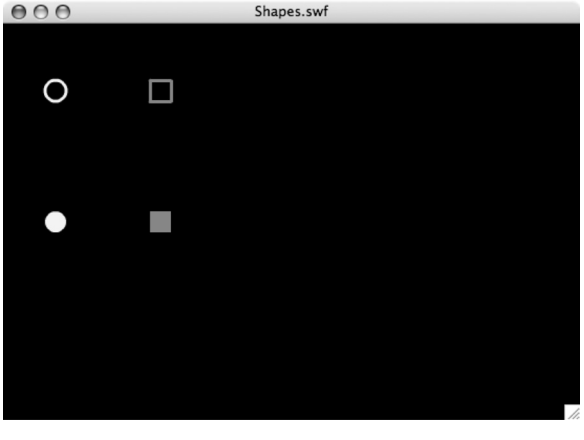


***Рис. 2.8.** Снимок окна сцены приложения Shapes*

Окно проекта нашего приложения, изображенное на рис. 2.9, показывает нам структуру нашего примера `Shapes`.

Классы продуктов: элементы образов

Фигуры на сцене (рис. 2.9) являются в данном примере продуктами. Их классы расширяют абстрактный класс `ShapeWidget` (пример 2.21). В отличие от предыдущих примеров мы определим `ShapeWidget` как аб-

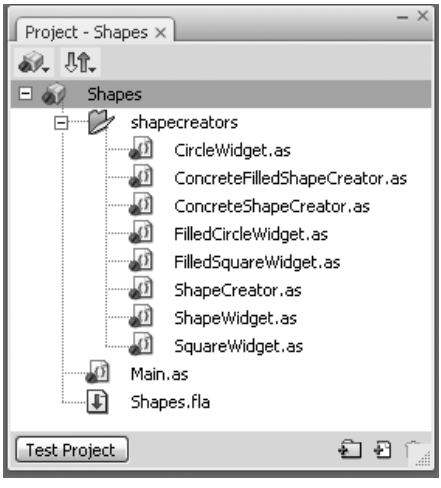


***Рис. 2.9.** Окно проекта для примера Shapes*

страктный интерфейс, потому что мы хотим сразу имплементировать поведение по умолчанию для всех фигур. Кроме того, `ShapeWidget` будет подклассом `Sprite`. Поведение по умолчанию определяется в методе `setLoc()`, который устанавливает координаты X и Y для спрайта. В `ShapeWidget` будет также определен абстрактный метод с именем `drawWidget()`, который будет рисовать спрайт.

Пример 2.21. ShapeWidget.as

```
package shapecreators
{
    import flash.display.Sprite;

    // Абстрактный класс (должен быть наследован, а не реализован)
    internal class ShapeWidget extends Sprite
    {
        // Абстрактный метод (должен быть замещен в подклассе)
        internal function drawWidget():void {}

        internal function setLoc(xLoc:int, yLoc:int):void {
            this.x = xLoc;
            this.y = yLoc;
        }
    }
}
```

Каждый объект `Shape`, `Sprite` и `MovieClip` имеет свойство `graphics`, являющееся объектом класса `Graphics`. Этот класс включает в себя свойства и методы для рисования и манипулирования линиями и фигурами, включая цвета, заполнение контуров и шаблоны. Конкретные классы,

порожденные от ShapeWidget, приведены в примерах с 2.22 по 2.25. Они рисуют соответствующие фигуры в своем *конструкторе*, используя методы, доступные через свойство graphics.

Пример 2.22. SquareWidget.as

```
package shapecreators {

    internal class SquareWidget extends ShapeWidget {

        override internal function drawWidget():void
        {
            graphics.lineStyle(3, 0xFF00FF);
            graphics.drawRect(-10, -10, 20, 20);
        }
    }
}
```

Пример 2.23. CircleWidget.as

```
package shapecreators {

    internal class CircleWidget extends ShapeWidget {

        override internal function drawWidget():void
        {
            graphics.lineStyle(3, 0xFFFF00);
            graphics.drawCircle(0, 0, 10);
        }
    }
}
```

Пример 2.24. FilledSquareWidget.as

```
package shapecreators {

    internal class FilledSquareWidget extends ShapeWidget {

        override internal function drawWidget():void
        {
            graphics.beginFill(0xFF00FF);
            graphics.drawRect(-10, -10, 20, 20);
            graphics.endFill();
        }
    }
}
```

Пример 2.25. FilledCircleWidget.as

```
package shapecreators {

    internal class FilledCircleWidget extends ShapeWidget {
```

```
        override internal function drawWidget():void
        {
            graphics.beginFill(0xFFFF00);
            graphics.drawCircle(0, 0, 10);
            graphics.endFill();
        }
    }
}
```

Классы создателей: создатели фигур

Класс `ShapeCreator`**, показанный в примере 2.26, определяет абстрактный интерфейс для классов создателей. Публично доступный метод** `draw()` **является для него настоящей рабочей лошадкой. Он вызывает фабричный метод** `createShape()` **и работает с возвращенным ему объектом** `ShapeWidget`**. Параметризованный фабричный метод** `createShape()` **должен рассматриваться как абстрактный метод и должен быть замещен в конкретном классе создателя.**

Пример 2.26. ShapeCreator.as

```
package shapecreators
{
    import flash.display.DisplayObjectContainer;
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class ShapeCreator
    {
        public function draw(cType:uint, target:DisplayObjectContainer,
                             xLoc:int, yLoc:int):void {
            var  shape = this.createShape(cType);
            shape.drawWidget();
            shape.setLoc(xLoc, yLoc); // установить положение по x и y
            target.addChild(shape); // добавить спрайт в список отображения
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        protected function createShape(cType:uint):ShapeWidget
        {
            throw new IllegalOperationError("Abstract method:
                                            must be overridden in a subclass");
            return null;
        }
    }
}
```

У метода `draw()` **имеется четыре параметра: вид создаваемой фигуры, объект сцены и координаты фигуры X и Y. Хотя классы создателей и рисуют фигуры на сцене, последние остаются невидимыми до тех**

пор, пока они не будут добавлены в *список отображения* для данной сцены с помощью метода `addChild()`. Клиенты передадут представителей класса `Stage` для обеспечения возможности нарисовать на сцене фигуры. Конкретные классы создателей `UnfilledShapeCreator` (пример 2.27) и `FilledShapeCreator` (пример 2.28) расширяют собой класс `ShapeCreator` (пример 2.26) и имплементируют фабричный метод `createShape`.

Пример 2.27. UnfilledShapeCreator.as

```
package shapecreators
{
    public class UnfilledShapeCreator extends ShapeCreator
    {
        public static const CIRCLE        :uint =  0;
        public static const SQUARE        :uint =  1;

        override protected function createShape(cType:uint):ShapeWidget
        {
            if (cType == CIRCLE)
            {
                trace("Creating new circle shape");
                return new CircleWidget();
            } else if (cType == SQUARE) {
                trace("Creating new square shape");
                return new SquareWidget();
            } else {
                throw new Error("Invalid kind of shape specified");
                return null;
            }
        }
    }
}
```

Пример 2.28. FilledShapeCreator.as

```
package shapecreators
{
    public class FilledShapeCreator extends ShapeCreator
    {
        public static const CIRCLE        :uint =  0;
        public static const SQUARE        :uint =  1;

        override protected function createShape(cType:uint):ShapeWidget
        {
            if (cType == CIRCLE)
            {
                trace("Creating new filled circle shape");
                return new FilledCircleWidget();
            } else if (cType == SQUARE) {
                trace("Creating new filled square shape");
                return new FilledSquareWidget();
```

```
            } else {
                throw new Error("Invalid kind of shape specified");
                return null;
            }
        }
    }
}
```

Клиенты

Заметьте, что когда клиент вызывает класс создателя, он должен передать в качестве параметра для его метода `draw()` представителя `Stage` («сцена»). Если ваш клиент является классом документа Flash, то у него должен быть доступ к сцене через свойство `this.stage`. В ActionScript 3.0 сцена не является глобально доступным объектом. К ней имеют доступ только те объекты, которые уже включены в список отображения.

```
// реализация конкретных создателей форм
var unfilledShapeCreator:ShapeCreator = new UnfilledShapeCreator();
var filledShapeCreator:ShapeCreator = new FilledShapeCreator();

// рисование незаполненных форм
unfilledShapeCreator.draw(UnfilledShapeCreator.CIRCLE, this.stage, 50, 75);
unfilledShapeCreator.draw(UnfilledShapeCreator.SQUARE, this.stage, 150, 75);

// рисование заполненных форм
filledShapeCreator.draw(FilledShapeCreator.CIRCLE, this.stage, 50, 200);
filledShapeCreator.draw(FilledShapeCreator.SQUARE, this.stage, 150, 200);
```

Как и в предыдущем примере, вы можете без особых усилий расширить функциональность этого приложения, добавив в него возможности по рисованию различных спрайтов с разнообразным поведением. Мы используем эту фабрику спрайтов в нашем заключительном примере этой главы – вертикальной стрелковой игре.

Пример: вертикальная стрелковая игра

Следующее приложение продемонстрирует нам полезность шаблона Фабричный метод при разработке динамичной игры; на экране в ответ на действия пользователя будет создаваться множество спрайтов. Когда вам заранее неизвестно, какие именно объекты придется создавать приложению во время игры, в вашей голове должна автоматически зажигаться лампочка напротив надписи «шаблон Фабричный метод». Мы разработаем часть вертикальной стрелковой игры в лучших традициях «Космических захватчиков». Наша игра будет основываться на рассмотренном примере фабрики спрайтов, поскольку все динамические объекты, которые будут появляться у нас на экране, будут производными классами от класса `Sprite`. Это будет не полная игра, а только

те ее части, которые покажут нам полезность использования шаблона Фабричный метод, такие как создание различных космических кораблей, включая различные снаряды, которые корабли будут запускать друг в друга.

Наша игра будет состоять из одного корабля героя, расположенного внизу сцены (рис. 2.10) и перемещаемого горизонтально с помощью мыши. Пять вражеских кораблей пришельцев будут расположены в ряд вверху сцены. Корабли пришельцев будут выпускать ядра из пушек (незаполненные кружочки) и мины пришельцев (незаполненные вращающиеся квадратики). Нажатие кнопки мыши будет приводить к выстрелу ядром из пушки героя (заполненным кружочком). В этом примере мы не будем имплементировать проверку коллизий (также известную как проверка попаданий) для определения попадания всех этих снарядов в космические корабли.

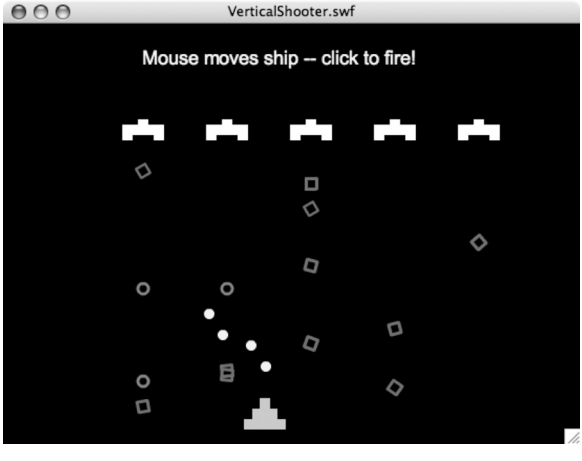


Рис. 2.10. *Вид окна вертикальной стрелковой игры с кораблями и снарядами*

Рисунок 2.11 показывает нам окно проекта для нашего примера. Оно состоит из двух пакетов, которые инкапсулируют в себе соответственно космические корабли и их вооружение. Пакет `weapons` будет использовать шаблон Фабричный метод и содержать в себе классы создателей (платформы вооружений), которые будут производить (выстреливать) различные снаряды. Пакет `ships` будет использовать разновидность фабричного метода для создания различных космических кораблей.

Классы продуктов

В нашем примере есть два различных вида продуктов: вооружения и космические корабли. Каждый продукт представляется некоторым спрайтом на сцене приложения.

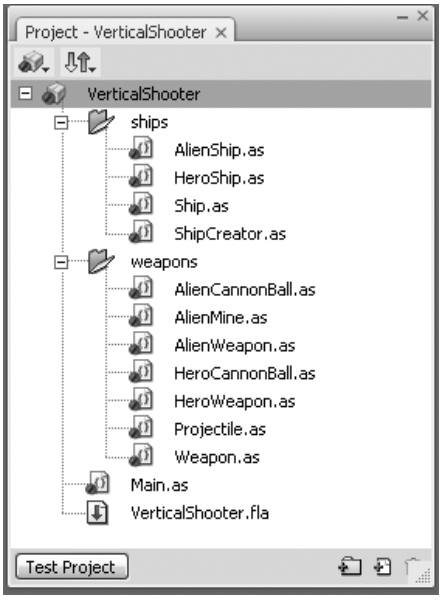


***Рис. 2.11.** Окно проекта вертикальной стрелковой игры*

Вооружения

Класс `Projectile`, показанный в примере 2.29, определяет собой абстрактный интерфейс для конкретных классов снарядов. Он должен рассматриваться как абстрактный класс и быть подклассом класса `Sprite`. В этом классе определяется свойство `nSpeed`, которое содержит скорость снаряда в пикселах на кадр. Его метод `drawProjectile()` должен рассматриваться как абстрактный и имплементироваться в подклассах. В этом классе определяется еще несколько методов с имплементацией по умолчанию. Метод `setLoc()` устанавливает координаты `X` и `Y` для спрайта. Метод `arm()` устанавливает скорость снаряда по умолчанию в 5 пикселов на кадр. Метод `release()` добавляет обработчик события `EnterFrame`, называемый `doMoveProjectile()`, который будет перемещать наш снаряд вертикально на `nSpeed` пикселов при каждом событии `EnterFrame`. Метод `doMoveProjectile()` осуществляет также важную функцию проверки, не вышел ли наш снаряд за верхнюю или нижнюю границу сцены. И если вышел, то он уберет данный спрайт из списка подписчиков на событие `ENTER_FRAME`, а также из списка отображения сцены. Поскольку после этого на данный объект снаряда не останется действующих ссылок, сборщик мусора Flash произведет освобождение памяти, которую этот объект занимал.

Пример 2.29. Projectile.as

```
package weapons
{
```

```
        import flash.display.Sprite;
        import flash.events.*;

        // Абстрактный класс (должен быть наследован, а не реализован)
        internal class Projectile extends Sprite
        {
            internal var nSpeed:Number // содержит скорость снаряда

            // Абстрактный метод (должен быть замещен в подклассе)
            internal function drawProjectile():void {}

            internal function arm():void
            {
                // Устанавливает скорость снаряда по умолчанию (5 пикселов в кадр)
                nSpeed = 5;
            }

            internal function release():void
            {
                // Подключить функцию doMoveProjectile на событие ENTER_FRAME
                this.addEventListener(Event.ENTER_FRAME, this.doMoveProjectile);
            }

            internal function setLoc(xLoc:int, yLoc:int):void
            {
                this.x = xLoc;
                this.y = yLoc;
            }

            // обновить спрайт снаряда
            internal function doMoveProjectile(event:Event):void
            {
                this.y += nSpeed; // переместить снаряд
                // убрать снаряд, если он вышел за границы сцены
                if ((this.y < 0) || (this.y > this.stage.stageHeight))
                {
                    // убрать обработчик события
                    this.removeEventListener(Event.ENTER_FRAME,
                                             this.doMoveProjectile);
                    this.stage.removeChild(this); // убрать спрайт со сцены
                }
            }
        }
    }
```

Поведение по умолчанию, определенное в классе `Projectile`**, подходит для большинства производных от него конкретных классов снарядов (примеры с 2.30 по 2.32). Это сильно упрощает данные классы. Им остается только научиться рисовать свой снаряд путем замещения и имплементации метода** `drawProjectile()`**. Метод** `arm()` **замещается в них для установки другой скорости снаряда. Здесь стоит отметить преимуще-**

ство использования абстрактных классов: код, необходимый для добавления новых снарядов, минимален, поскольку в большинстве случаев желаемое поведение просто совпадает с наследуемым поведением по умолчанию.

Пример 2.30. HeroCannonBall.as

```
package weapons {

    internal class HeroCannonBall extends Projectile {

        override internal function drawProjectile():void
        {
            graphics.beginFill(0xFFFF00);
            graphics.drawCircle(0, 0, 5);
            graphics.endFill();
        }

        override internal function arm():void {
            nSpeed = -10; // установить скорость
        }
    }
}
```

Пример 2.31. AlienCannonBall.as

```
package weapons {

    internal class AlienCannonBall extends Projectile {

        override internal function drawProjectile():void
        {
            graphics.lineStyle(3, 0xFF00FF);
            graphics.drawCircle(0, 0, 5);
        }

        override internal function arm():void {
            nSpeed = 8; // установить скорость
        }
    }
}
```

Пример 2.32. AlienMine.as

```
package weapons {

    import flash.events.*;


    internal class AlienMine extends Projectile {

        override internal function drawProjectile():void
```

```
        {
            graphics.lineStyle(3, 0xFF0000);
            graphics.drawRect(-5, -5, 10, 10);
        }

        override internal function arm():void {
            nSpeed = 2; // установить скорость
        }

        override internal function doMoveProjectile(event:Event):void {
            super.doMoveProjectile(event);
            this.rotation += 5; // повернуть
        }
    }
}
```

Все снаряды рисуются с помощью методов класса `Graphics`, доступного через свойство `graphics`. Скорость снаряда для класса `HeroCannonBall` («шарик из пушки героя») устанавливается в отрицательное значение, поскольку он движется снизу вверх на нашей сцене (пример 2.30). Между тем снаряды пришельцев двигаются сверху вниз (см. расположение космических кораблей на рис. 2.10). Кроме того, обратите внимание на замещенный метод `doMoveProjectile()` в классе `AlienMine`. Он вызывает поведение по умолчанию в суперклассе с помощью свойства `super`, но добавляет к нему команды, которые заставляют спрайт поворачиваться. Поэтому мины пришельцев медленно вращаются по мере своего продвижения вниз.

Космические корабли

Класс `Ship`, приведенный в примере 2.33, определяет собой абстрактный интерфейс для конкретных классов космических кораблей. Класс `Ship` определен как подкласс класса `Sprite` и должен рассматриваться как абстрактный класс. В нем определяется метод `setLoc()`, который устанавливает координаты X и Y спрайта космического корабля. В нем также определяются два метода без имплементации, которые должны рассматриваться как *абстрактные* методы. Метод `drawShip()` должен быть замещен и имплементирован, чтобы рисовать корабль. Метод `initShip()` тоже должен быть замещен, чтобы инициализировать поведение корабля, в частности добавить обработчики нужных событий.

Пример 2.33. Ship.as

```
package ships
{
    import flash.display.Sprite;
    import flash.events.*;

    // Абстрактный класс (должен быть наследован, а не реализован)
    internal class Ship extends Sprite
```

```
    {
        internal function setLoc(xLoc:int, yLoc:int):void
        {
            this.x = xLoc;
            this.y = yLoc;
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        internal function drawShip():void {}

        // Абстрактный метод (должен быть замещен в подклассе)
        internal function initShip():void {}
    }
}
```

Классы `AlienShip` **(«корабль пришельцев») (пример 2.34) и** `HeroShip` **(«корабль героя») (пример 2.35) расширяют класс** `Ship` **(пример 2.33).**

Пример 2.34. AlienShip.as

```
package ships
{
    import flash.display.*;
    import flash.events.*;
    import weapons.AlienWeapon;

    public class AlienShip extends Ship
    {
        private var weapon:AlienWeapon;
        // доступные снаряды
        private const aProjectiles:Array = [AlienWeapon.CANNON, AlienWeapon.MINE];

        override internal function drawShip():void
        {
            graphics.beginFill(0xFFFFFF); // белый цвет
            graphics.drawRect(-5, -10, 10, 5);
            graphics.drawRect(-20, -5, 40, 10);
            graphics.drawRect(-20, 5, 10, 5);
            graphics.drawRect(10, 5, 10, 5);
            graphics.endFill();
        }

        override internal function initShip():void
        {
            // реализуем создатель снарядов пришельцев
            weapon = new AlienWeapon();
            // добавим метод doFire() этого объекта
            // как обработчик события ENTER_FRAME
            this.stage.addEventListener(Event.ENTER_FRAME, this.doFire);
        }

        protected function doFire(event:Event):void
```

```
        {
            // стреляем случайным образом (шанс выстрела – 4% каждый кадр)
            if (Math.ceil(Math.random() * 25) == 1)
            {
                // выбираем случайный снаряд
                var cProjectile:uint =
                  aProjectiles[Math.floor(Math.random() * aProjectiles.length)];
                weapon.fire(cProjectile, this.stage, this.x, this.y + 15);
            }
        }
    }
}
```

Пример 2.35. HeroShip.as

```
package ships
{
    import flash.display.*;
    import weapons.HeroWeapon;
    import flash.events.*;

    internal class HeroShip extends Ship
    {
        private var weapon:HeroWeapon;

        override internal function drawShip():void
        {
            graphics.beginFill(0x00FF00); // зеленый цвет
            graphics.drawRect(-5, -15, 10, 10);
            graphics.drawRect(-12, -5, 24, 10);
            graphics.drawRect(-20, 5, 40, 10);
            graphics.endFill();
        }

        override internal function initShip():void
        {
            // реализуем создатель снарядов героя
            weapon = new HeroWeapon();
            // добавляем методы doMoveShip() и doFire()
            // на события MOUSE_MOVE и MOUSE_DOWN сцены
            this.stage.addEventListener(MouseEvent.MOUSE_MOVE, this.doMoveShip);
            this.stage.addEventListener(MouseEvent.MOUSE_DOWN, this.doFire);
        }

        protected function doMoveShip(event:MouseEvent):void
        {
            // установить координату x спрайта
            // соответственно относительному положению мыши
            this.x = event.stageX;
            event.updateAfterEvent(); // обработать это событие в первую очередь
        }
```

```
        protected function doFire(event:MouseEvent):void
        {
            weapon.fire(HeroWeapon.CANNON, this.stage, this.x, this.y - 25);
            event.updateAfterEvent(); // обработать это событие в первую очередь
        }
    }
}
```

Обратите внимание, что методы `initShip()` в обоих производных классах устанавливают обработчики различных перехватываемых событий. Обработчик событий для класса `HeroShip` реагирует на события `MOUSE_MOVE` и `MOUSE_DOWN`, посылаемые сцене. Перехват этих событий для всей сцены необходим нам потому, что корабль героя должен отвечать на события мыши, даже если ее фокус в этот момент не направлен непосредственно на его спрайт. Кроме того, метод `initShip()` инициализирует соответствующий класс создателя снарядов. Обработчик события `ENTER_FRAME` класса `AlienShip` является методом `doFire()`, который выстреливает случайные снаряды из списка доступных для него, используя классы их создателей.

Классы создателей

Мы заканчиваем этот пример рассмотрением двух видов классов создателей, соответствующих двум типам продуктов. Один класс создателей инкапсулирует в себе создание снарядов, а другой – создание космических кораблей.

Вооружение

Класс `Weapon` (пример 2.36) является абстрактным интерфейсом, который инкапсулирует в себе создание вооружения. Его лучше описать как некоторую платформу вооружений, которая может выпускать различные виды снарядов. Его публично доступный метод `fire()` обращается к фабричному методу `createProjectile()`.

Пример 2.36. Weapon.as

```
package weapons
{
    import flash.display.Stage;
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class Weapon
    {

        public function fire(cWeapon:uint, target:Stage, xLoc:int, yLoc:int):void
        {
            var projectile:Projectile = this.createProjectile(cWeapon);
            trace("Firing " + projectile.toString());
```

```
            // рисуем снаряд
            projectile.drawProjectile();
            // устанавливаем стартовые координаты x и y
            projectile.setLoc(xLoc, yLoc);
            // вооружаем снаряд (замещаем значение скорости,
               установленное по умолчанию)
            projectile.arm();
            // добавляем снаряд в список отображения
            target.addChild(projectile);
            // заставляем снаряд двигаться, добавляя обработчик события enterframe
            projectile.release();
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        protected function createProjectile(cWeapon:uint):Projectile
        {
            throw new IllegalOperationError("Abstract method:
                                            must be overridden in a subclass");
            return null;
        }
    }
}
```

Классы AlienWeapon **(пример 2.37) и** HeroWeapon **(пример 2.38) расширяют класс** Weapon **(пример 2.36) и имплементируют фабричный метод** createProjectile()**.**

Пример 2.37. AlienWeapon.as

```
package weapons
{
    public class AlienWeapon extends Weapon
    {
        public static const CANNON    :uint =  0;
        public static const MINE        :uint =  1;

        override protected function createProjectile(cWeapon:uint):Projectile
        {
            if (cWeapon == CANNON)
            {
                trace("Creating new alien cannonball");
                return new AlienCannonBall();
            } else if (cWeapon == MINE) {
                trace("Creating new alien mine");
                return new AlienMine();
            } else {
                throw new Error("Invalid kind of projectile specified");
                return null;
            }
        }
    }
}
```

Пример 2.38. HeroWeapon.as

```
package weapons
{
    public class HeroWeapon extends Weapon
    {
        public static const CANNON    :uint =  0;

        override protected function createProjectile(cWeapon:uint):Projectile
        {
            if (cWeapon == CANNON)
            {
                trace("Creating new Hero cannonball");
                return new HeroCannonBall();
            } else {
                throw new Error("Invalid kind of projectile specified");
                return null;
            }
        }
    }
}
```

Создатель корабля

Конкретный класс создателя корабля `ShipCreator` (пример 2.39) инкапсулирует в себе процесс его создания. Нам здесь не нужно полностью инкапсулировать информацию о корабле героя и корабле пришельцев. В конце концов, у нас только один корабль героя и один вид кораблей пришельцев.

Пример 2.39. ShipCreator.as

```
package ships
{
    import flash.display.Stage;

    public class ShipCreator
    {
        public static const HERO   :uint =  0;
        public static const ALIEN  :uint =  1;

        public function addShip(cShipType:uint, target:Stage, xLoc:int,
                                yLoc:int):void
        {
            var ship:Ship = this.createShip(cShipType);
            ship.drawShip(); // рисуем корабль
            ship.setLoc(xLoc, yLoc); // устанавливаем координаты x и y
            target.addChild(ship); // добавляем спрайт на сцену
            ship.initShip(); // инициализируем корабль
        }

        private function createShip(cShipType:uint):Ship
```

```
        {
            if (cShipType == HERO)
            {
                trace("Creating new hero ship");
                return new HeroShip();
            } else if (cShipType == ALIEN) {
                trace("Creating new alien ship");
                return new AlienShip();
            } else {
                throw new Error("Invalid kind of ship specified");
                return null;
            }
        }
    }
}
```

Конкретные классы создателей

До рассмотрения класса `ShipCreator` (в примере 2.29) все наши примеры классов создателей были абстрактными. Конкретные классы создателей полностью имплементируют фабричный метод, а не оставляют имплементацию своим подклассам. Таким образом, добавление нового продукта требует изменения существующего фабричного метода в конкретном классе создателя. Подобная переделка кода является не таким изящным решением, как расширение абстрактного класса при реализации необходимых добавлений в приложение. Конкретные классы создателей полезны для отделения классов продуктов от клиентов, которые будут их использовать. Но если к этому еще добавляется перспектива изменения требований к классам, то выбор в пользу абстрактных классов создателей более разумен в большинстве случаев.

Клиенты

У нас есть множество клиентов, использующих класс создателя `ShipCreator` (из примера 2.39), который позволяет им размещать космические корабли на сцене.

```
// реализуем создатель кораблей
var shipFactory:ShipCreator = new ShipCreator();

// ставим корабль героя
shipFactory.addShip(ShipCreator.HERO, this.stage,
                this.stage.stageWidth / 2, this.stage.stageHeight - 20);
// ставим корабли пришельцев
for (var i:Number = 0; i < 5; i++)
{
    shipFactory.addShip(ShipCreator.ALIEN, this.stage,
        120 + 80 * i, 100);
}
```

Кроме того, космические корабли героя и пришельцев обращаются к своим соответствующим классам вооружений, чтобы создавать и выстреливать свои снаряды. Таким образом, космические корабли являются клиентами классов снарядов.

Заключение

При разработке программного обеспечения перемены в нем неизбежны. Требования к приложениям меняются, и иногда для воплощения не предвиденных в исходном проекте программы изменений используются «грязные трюки». Противоядием для данной распространенной проблемы является надежное изначальное проектирование приложения, которое будет способно перенести предстоящие изменения и модификации. А лучший способ справиться с ними – управление зависимостями между сегментами кода. Рассмотрим пример, когда класс `Client` создает нового представителя класса `Product`, используя ключевое слово `new`, и сохраняет получившийся объект в переменной. Это очень распространенная практика, которая создает жесткую зависимость между двумя указанными классами. Ее называют сильной или *жесткой связью*. Изменения в одном из таких классов с большой вероятностью распространятся и на другой класс. Шаблон Фабричный метод представляет собой отличное средство для управления подобными зависимостями, поскольку он воздвигает разделительную стену между такими классами. Данный шаблон не исключает совсем зависимость между ними, он задает некоторые ее рамки, которые позволяют ею управлять.

3

Шаблон Одиночка

Самые важные качества вещей обычно
скрыты от нас ввиду их простоты и обыденности.

Людвиг Витгенштейн

Превыше всего служи единственной цели.
Имей приемлемое и практичное предназначение
и посвяти себя ему безраздельно.

Джеймс Аллен

Что такое шаблон Одиночка?

Этот шаблон вы скорее всего уже использовали даже не замечая этого. Например, если у вас есть некоторый класс, который хранит общее число очков в вашей игре, то вы пожелаете, чтобы в ней существовал только один его представитель. Нет никакого смысла в двух таких хранителях счета, когда во всей игре есть только один счет, как в игре пинбол для одного игрока. Аналогичным образом, если вы создаете музыкальное приложение, вы захотите, чтобы оно играло только одну мелодию в каждый момент времени, и вам потребуется единственный представитель класса, который будет играть музыку. Фактически почти в каждом приложении имеются некоторые элементы, которые должны существовать в нем в единственном экземпляре, и именно здесь вам будет удобно использовать шаблон проектирования Одиночка.

Ключевые возможности

Кратко говоря, у Одиночки есть две ключевые возможности:

1. Только один представитель его класса может быть создан в приложении.
2. У его класса имеется единственная глобальная точка доступа.

Вы можете подумать: «А разве это сложно? Как разработчик я могу просто создать только одного представителя класса и использовать для его хранения глобальную переменную. Все, проблема решена! Перейду-ка я к изучению следующего шаблона».

Во-первых, как вы видели в главе 1, ООП и шаблоны проектирования были задуманы для команд разработчиков, а не для одного программиста, работающего независимо от других. Поэтому если вы работаете в команде, шаблон Одиночка может быть использован для того, чтобы гарантировать, что другой программист не создаст представителя класса, когда вы это уже сделали. Шаблон проектирования Одиночка просто запретит другим разработчикам создавать более одного представителя, выделенного для данного шаблона класса.

Во-вторых, использование глобальных переменных никак не ограничивает их использование только одним разом. Они могут появляться откуда угодно, от пакетов верхнего уровня до отдельных функций внутри классов. Глобальный доступ должен идти рука об руку с контролем создания таких объектов.

Мы не хотим этим сказать, что создание класса Одиночки является трудным делом. Вы можете создать его, используя отдельный файл, и в базовом классе не будет много кода. Вы можете добавлять в него любые методы и свойства, какие только захотите, и шаблон Одиночка не будет ограничивать вас в этом. Главной задачей, которую вы должны хорошенько понять, является гарантия того, что работа будет происходить только с одним его представителем в любой момент времени.

Модель Одиночки

Если мы посмотрим на диаграмму классов шаблона Одиночка, мы не обнаружим на ней ни абстрактных классов, ни интерфейсов. Ключевыми моментами здесь являются использование статического метода `Instance()` и связь с имплементационным псевдокодом. Как вы увидите, этот шаблон использует статические переменные и методы класса для имплементации глобального доступа и уникальности представителя класса. Рисунок 3.1 показывает эту простую, но мощную структуру.

Перед тем как перейти от диаграммы к работающей модели, нам необходимо разобрать несколько тем ООП применительно к ActionScript 3.0. Главным образом нам надо поговорить о том, как создавать самона-

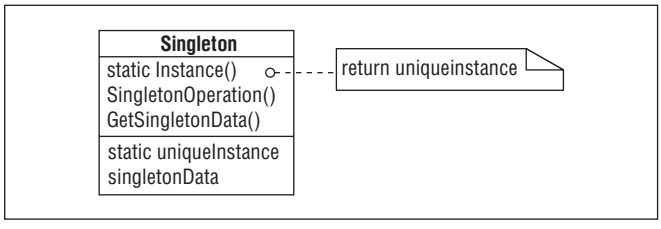


Рис. 3.1. *Структура Одиночки*

страиваемые классы. Как и в большинстве других шаблонов, отдельные элементы Одиночки работают с той частью шаблона, которая *имплементирует* или *расширяет* другие его части. Так что нам нужно вернуться к основам.

Ключевые концепции ООП, используемые в шаблоне Одиночка

Первой темой, которую нам надо обсудить, является использование приватных функций в качестве конструкторов. При имплементации шаблона Одиночка на языках ActionScript 2.0 и Java «классический» шаблон Одиночка создается с помощью приватного конструктора, так что при соблюдении определенной схемы работы такого конструктора только один представитель контролируемого им класса может быть создан в программе. Пример 3.1 показывает такую имплементацию:

Пример 3.1. Классический Одиночка

```
class Singleton
{
    private static var instance:Singleton;
    private function Singleton()
    {
    }
    public static function getInstance():Singleton
    {
        if (instance == null) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}
```

Все это выглядит очень просто. Однако посмотрите внимательно, и вы обнаружите много существенных моментов. Давайте пройдемся по ключевым точкам:

Приватная статическая переменная типа Singleton

Свойство приватности данной переменной говорит о том, что вы не можете к ней обратиться извне данного класса. А ее статичность означает, что вы можете с ней работать методами данного класса, предназначенными для инициации объекта `Singleton`.

Приватный конструктор класса

Пример 3.1 очень просто реализуется в ActionScript 2.0. Но в ActionScript 3.0 мы сталкиваемся с проблемой, состоящей в том, что приватные функции не могут быть использованы в качестве конструкторов в пакетах. Чтобы создавать представителей классов, конструкторы должны быть публичными и являться частью пакета. Мы обсудим эту тему подробнее позже, когда будем создавать эквивалент приватного конструктора класса.

Публичная статическая функция для создания представителя класса

Метод `getInstance()` удивительно прост. Он проверяет, не создан ли уже представитель данного класса, и, если нет, создает его. В противном случае он просто возвращает уже имеющегося представителя класса.

Простота шаблона Одиночка станет более очевидной после рассмотрения его работы на конкретном примере. Однако при работе с ActionScript 3.0 мы не можем использовать приватные конструкторы классов. В следующем разделе мы обсудим обходной путь для этой проблемы.

Создание и использование приватных конструкторов классов

Вместо того чтобы демонстрировать, как ваш код выдает ошибку при попытке создания приватного конструктора внутри пакета, мы лучше покажем, как эффективно создавать ваш собственный приватный класс и использовать его в качестве конструктора. Фактически мы имплементируем его внутри одного файла сценария.

Чтобы создать класс, нам нужно создать его публичный конструктор в пакете. Поскольку мы не можем создавать приватный класс внутри пакета, нам придется создать его вне пакета, а потом каким-то образом заполучить его для работы внутри пакета. Для этого мы используем размещенный в отдельном *as*-файле сценарий, который сначала создает публичный класс, закрывает пакет, а затем создает приватный класс вне пакета. Приватный класс вызывается тогда внутри публичного класса. Пример 3.2 показывает, как это сделать:

Пример 3.2. PublicClass.as

```
1 package
2 {
```

```
 3     import flash.display.Sprite;
 4     public class PublicClass extends Sprite
 5     {
 6         public function PublicClass()
 7         {
 8             PrivateClass.alert();
 9         }
10     }
11 }
12 // Конец пакета
13 //**************
14 // Приватный класс
15 class PrivateClass
16 {
17     public function PrivateClass(){
18     }
19
20      public static function alert():void
21     {
22         trace("This is from a private class");
23     }
24 }
```

Сохраните этот файл как *PublicClass.as*. Создайте *fla*-файл с именем *TestPrivate.fla* и в его поле класс документа введите `PublicClass`. Когда вы его протестируете, вы увидите на выходе:

```
This is from a private class
```

Заметьте, что в строке 8 приватный класс `PrivateClass` вызывается просто указанием его имени и метода. Никаких конструкторов при этом не используется.

Создание представителя приватного класса

Не забывая о способе имплементации в примере 3.2, нам нужно разработать конструктор класса, который будет работать как приватный. Если приватный класс сможет быть частью метода создания представителя (`getInstance()`), тогда мы сможем действовать точно так же, как и с приватным конструктором класса.

Почему в ActionScript 3.0 нет приватных классов? Приватные классы до сих пор не введены в ECMAScript, и ActionScript 3.0 следует этому стандарту. Из-за нескольких нетривиальных проблем и ECMAScript, и ActionScript 3.0 придется немного подождать появления в них приватных классов. Другие варианты заключались в том, чтобы либо отложить выход ActionScript 3.0, либо оставить реализацию приватных классов только обходными путями.

Рассмотрим пример 3.3:

Пример 3.3. Использование приватного класса в конструкторе публичного класса

```
package
{
    public class PublicClass
    {
        private static var instance:PublicClass;
        public function PublicClass(pvt:PrivateClass) {
        }
        public static function getInstance():PublicClass
        {
            if (instance == null) {
            PublicClass.instance=new PublicClass (new PrivateClass());
        }
        return PublicClass.instance;
        }
    }
}
class PrivateClass
{
    public function PrivateClass() {
        trace("Private class is up");
    }
}
```

Ключевым моментом является то, что приватный класс передается как параметр конструктору публичного класса. Потом, когда создается новый представитель с помощью метода `getInstance()`, сюда включается параметром новый представитель приватного класса.

Минималистский абстрактный Одиночка

Чтобы проиллюстрировать шаблон проектирования Одиночка самым простым способом, мы придерживаемся в примере 3.4 схемы реализации классического Одиночки и вносим в него минимальные изменения. Добавленное выражение `else` показывает, что произойдет, если случится попытка создать более одного представителя этого класса. Позднее мы его удалим. Дополнительные команды `trace` в примере 3.4 используются для демонстрации того, как данный сценарий работает.

Пример 3.4. Singleton.as

```
package
{
    public class Singleton
    {
        private static var _instance:Singleton;
        public function Singleton(pvt:PrivateClass) {
```

```
        }
        public static function getInstance():Singleton
        {
            if(Singleton._instance == null)
            {
                Singleton._instance=new Singleton(new PrivateClass());
                trace("Singleton instantiated");
            }
            else
            {
                trace("Sorry--already have a Singleton instantiated")
            }
            return Singleton._instance;
        }
    }
}
class PrivateClass
{
    public function PrivateClass() {
        trace("Private class is up");
    }
}
```

Сохраните этот файл как *Singleton.as*. Теперь у нас есть Одиночка, с которым можно работать. Давайте же начнем его использовать.

Использование параметра приватного класса

Когда вы собираетесь создать представителя класса `Singleton` в программе, вы не можете использовать для этого следующий формат:

```
var mySingleton:Singleton = new Singleton( );
```

Данный формат сталкивается с ироничной проблемой необходимости аргумента для конструктора. Сейчас в строке описания ноль аргументов, а вам нужен один. Необходимым аргументом является ссылка на приватный класс, который доступен только через публичный класс, представителя которого вы и хотите создать. (В этом и состоит парадокс, если можно так сказать.) Посмотрите на неправильную имплементацию класса `Singelton` в примере 3.5:

Пример 3.5. BadImplementation.as

```
package
{
    import flash.display.Sprite
    public class BadImplementation extends Sprite
    {
        public function BadImplementation()
        {
            var mySingleton=new Singleton();
        }
```

```
        }
    }
```

Сохраните этот файл как *BadImplementation.as* в том же каталоге, где содержится *Singleton.as*. В свойствах документа **Flash в поле класса документа** введите `BadImplementation` и протестируйте это приложение. Как и предполагалось, вы получите ошибку компиляции, изображенную на рис. 3.2.

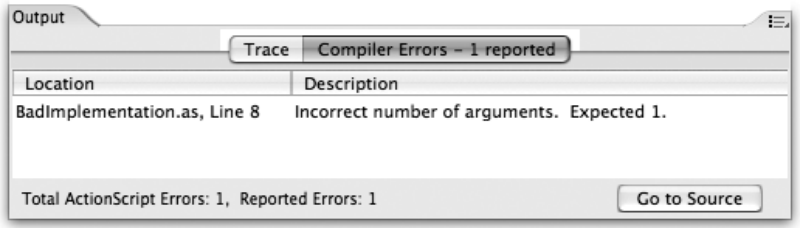


***Рис. 3.2.** Ошибка компиляции: ожидается аргумент*[1]

Как только вы увидите, что конструктор класса `Singelton` содержит в качестве аргумента класс `PrivateClass`, естественной реакцией будет изменить наш код и включить в него аргумент `PrivateClass`, как это показано в примере 3.6.

Пример 3.6. Попытка добавить приватному классу параметр

```
package
{
    import flash.display.Sprite
    public class BadImplementation extends Sprite
    {
        public function BadImplementation()
        {
            var mySingleton=new Singleton(new PrivateClass());
        }
    }
}
```

Сохраните заново файл *BadImplementation.as* и опять протестируйте его. На этот раз вы получите другую ошибку, показанную на рис. 3.3:

Что она означает? Компилятор не находит приватный класс с именем `PrivateClass`, поскольку он не помнит, чтобы вы его создавали, и не может его найти в пути поиска классов. Однако проблема в другом. Проблема состоит в том, что это приватный класс и к нему нет доступа иным путем, кроме как через класс `Singelton`, представителя которого вы пытаетесь создать.

[1] BadImplementation.as, строка 8: Неверное число аргументов. Ожидается 1. – *Прим. науч. ред.*

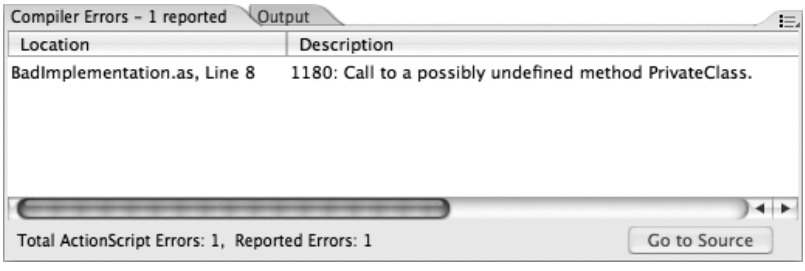


Рис. 3.3. *Компилятор не находит класс*[1]

Метод создания представителя Одиночки

При разработке класса `Singelton` нам пришлось включить специальный метод в его конструктор для создания его представителя. Как вы уже видели, стандартные методы, используемые для создания представителей классов, просто не действуют. Функция `getInstance()` решает эту проблему. Следующий сегмент кода показывает часть программы, которая позволяет получить доступ к приватному классу, необходимому для создания представителя его класса:

```
public static function getInstance():Singleton
{
    if(Singleton._instance == null)
    {
        Singleton._instance=new Singleton(new PrivateClass());
        trace("Singleton instantiated");
    }
.............
    return Singleton._instance
}
```

Приватная статическая переменная `_instance` будет содержать представителя класса `Singleton`. Поскольку нам нужно, чтобы был только один такой представитель, наш код проверяет, не был ли он уже создан. Любая ActionScript 3.0 пользовательская переменная сложного типа, которой не было присвоено значение, по умолчанию имеет значение `null`. Поэтому после того как мы проверим, что переменная `Singleton._instance` равна `null` (т. е. не было создано другого представителя), наш сценарий создаст его. А здесь мы видим строку, пытающуюся создать нового представителя вне данного класса:

```
....new Singleton(new PrivateClass());
```

Это возможно потому, что на `PrivateClass` имеется ссылка изнутри класса `Singleton`.

---

[2] BadImplementation.as, строка 8: 1180: Вызов возможно неопределенного метода `PrivateClass`. – *Прим. науч. ред.*

Теперь мы можем написать класс, который будет создавать представителя класса `Singleton`. Откройте новый файл ActionScript и введите в него код, показанный в примере 3.7. Сценарий пытается создать два представителя класса `Singleton`. Если наш класс задан корректно, он позволит вам создать только одного своего представителя.

Пример 3.7. Создание представителя класса Singleton

```
package
{
    import flash.display.Sprite;
    public class SingletonTest extends Sprite {
        public function SingletonTest() {
            var firstSingleton:Singleton = Singleton.getInstance();
            var secondSingleton:Singleton=Singleton.getInstance();
        }
    }
}
```

Сохраните этот файл как *SingletonTest.as*; откройте новый документ Flash и введите в его поле класс документа `SingletonTest`. Оба эти файла должны быть сохранены в том же каталоге, где содержится файл *Singleton.as*. Теперь протестируйте данное приложение. Рисунок 3.4 показывает то, что вы должны увидеть:

Рис. 3.4. *Создание одного представителя класса*

Тот факт, что вы видите сообщение: «Sorry--already have a Singleton instantiated» (Извините, Одиночку уже создали), говорит о том, что был успешно создан только один из двух представителей класса `Singleton`. Именно на это мы и надеялись. Если бы были созданы оба представителя, то нам пришлось бы вернуться к самому началу. Однако, после того как один из них был создан (что показали нам сообщения «Приватный класс создан» и «Одиночка создан»), создание второго такого представителя было заблокировано.

Один представитель и разные ссылки

Просмотрев классическую реализацию шаблона проектирования Одиночка, можно заметить, что независимо от результатов попытки создать нового представителя этого класса программа всегда доходит до точки,

где возвращается Singleton._instance. Мы знаем, что если при проверке будет обнаружено, что один представитель класса, Singleton._instance, уже существует, то второй такой создаваться не будет, а любой существующий представитель будет возвращен.

Однако, означает ли это, что только одна ссылка может быть ассоциирована с этим единственным представителем? Что произойдет, например, если сначала будет создан один представитель под одним именем, а потом мы попытаемся создать другого представителя с другим именем?

Чтобы убедиться, что единственный представитель может быть ассоциирован более чем с одной ссылкой (именем или меткой), нам надо немного изменить основной класс Singleton. В каталоге, отличном от каталога текущего класса Singleton, создайте новый файл *Singleton.as* и запишите в него сценарий из примера 3.8.

Пример 3.8. Singleton.as

```
package
{

    public class Singleton
    {
        private var _msg:String;
        private static var _instance:Singleton;
        public function Singleton(pvt:PrivateClass)
        {
        }
        public static function getInstance():Singleton
        {

            if(Singleton._instance == null)
            {
                Singleton._instance=new Singleton(new PrivateClass());
                trace("Singleton instantiated");
            }
            return Singleton._instance;
        }

        // Методы получателя и установщика

        public function getMsg():String
        {
            return _msg;
        }
        public function setMsg(alert:String):void
        {
            _msg = alert;
        }
    }
}
```

```
class PrivateClass
{
    public function PrivateClass() {
    trace("PrivateClass called");
    }
}
```

Этот класс `Singleton` имеет методы получателя и установщика. Если мы создадим двух различных представителей с разными именами и попытаемся установить им различные значения для переменной `msg`, мы должны будем увидеть только одно из них (последнее установленное) при ее опросе. Таким образом, у нас может быть любое число имен, ассоциированных с одним представителем, и все они будут прекрасно функционировать. Однако все они будут ссылаться только на одного представителя класса `Singleton`. Нижеследующий алгоритм показывает логику его работы:

- Метка А создает представителя класса `Singleton` (успешно).
- Метка А с помощью метода `set` устанавливает сообщение «Message A» (успешно).
- Метка Б создает представителя `Singleton` (возвращается уже созданный представитель).
- Метка Б с помощью метода `set` устанавливает сообщение «Message B» (успешно).
- Метка А с помощью метода `get` выводит установленное сообщение. (Что будет выведено?)

Чтобы ответить на этот вопрос, откройте новый файл ActionScript и добавьте в него код из примера 3.9.

Пример 3.9. SingletonTest.as

```
package
{
    import flash.display.Sprite;
    public class SingletonTest extends Sprite {
        public function SingletonTest() {

            var firstSingleton:Singleton = Singleton.getInstance();
            firstSingleton.setMsg("Singleton instance: firstSingleton");
            var secondSingleton:Singleton = Singleton.getInstance();
            secondSingleton.setMsg("Singleton instance: secondSingleton");

            trace(firstSingleton.getMsg());
            trace(secondSingleton.getMsg());

        }
    }
}
```

Сохраните этот файл под именем *SingletonTest.as* в том же каталоге, где содержится новый файл *Singleton.as*. Откройте новый документ Flash, введите в его поле класс документа `SingletonTest` и сохраните его *fla*-файл в том же каталоге, что и первые два файла ActionScript. Когда вы его протестируете, ваше окно вывода покажет, что существует только один представитель класса Одиночки, как это видно на рис. 3.5.

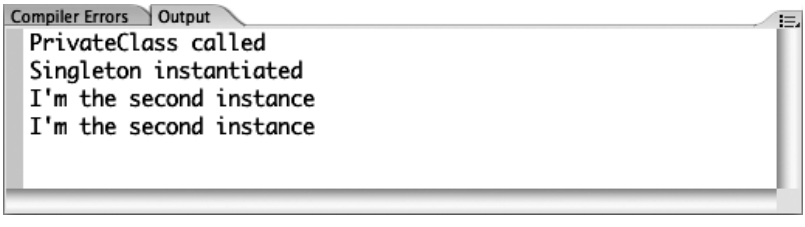


Рис. 3.5. *Две метки и один представитель*[1]

У вас, вероятно, возникнет вопрос, как такое вообще возможно, если при создании второго представителя мы на самом деле не попали в ту часть конструктора, которая занимается его созданием:

```
Singleton._instance=new Singleton(new PrivateClass( ));
```

Однако это было и не нужно из-за строки

```
return Singleton._instance
```

которая содержит уже созданного представителя `Singleton` и просто возвращает его для нового имени.

Пока вы можете гарантировать, что только один представитель класса `Singleton` будет существовать в любой момент времени в программе, на него может быть одновременно несколько ссылок. Для общих свойств такого класса, использующих инкременты или декременты для изменения своих значений, это может оказаться очень кстати, поскольку позволяет вам быть уверенными, что для всех операций будет использован только один источник данных. Если, например, вы создаете игру, в которой происходит спасение шахтеров, запертых обвалом в шахте, то вам нужно хранить оставшееся у них количество кислорода. Если вы создаете класс `Oxygen` как Одиночку, то все классы шахтеров могут создать на него ссылку, и каждый из них может через специальный свой метод уменьшать количество оставшегося воздуха. Поскольку это количество должно быть одинаковым для всех, шаблон Одиночка становится в данном случае удобным средством для работы с данными в одном классе, но для многих ссылок.

[1] У автора здесь ошибка. Как видно из примера 3.9, в окне вывода должно появиться «Singleton instance: secondSingleton» вместо «I'm the second instance». – *Прим. науч. ред.*

Когда использовать шаблон Одиночка

Как уже было сказано в начале этой главы, шаблон проектирования Одиночка лучше всего использовать, когда вам нужен один и только один представитель его класса в приложении. Мы рассмотрели пару примеров, но шаблон Одиночка настолько широко применяется в объектно-ориентированном программировании, что задавать поставленный вопрос, это все равно что спрашивать, когда нам использовать переменную.

Более того, разработчики часто комбинируют шаблон проектирования Одиночка с другими шаблонами. Там, где им требуется единственная, глобальная точка доступа к единственному представителю класса, можно часто увидеть шаблон Одиночка. Кроме того, несколько классов Одиночек может использоваться одновременно, когда у вас имеются разные объекты, но только единственный экземпляр каждого из них имеет право на существование в приложении.

Чтобы показать вам как можно шире и исчерпывающе все способы использования шаблона Одиночка, мы рассмотрим далее три примера. Первый из них, Сообщение, покажет нам, как только одно сообщение может быть выведено на сцену. Он представляет элементы интерфейса пользователя, которые выдают ему некоторую информацию, такие как окно диалога или подобные ему механизмы обратной связи. Только один такой элемент должен быть создан в приложении во избежание противоречивых сообщений пользователю. Данный пример также продемонстрирует, как подключить класс с отображаемым объектом на сцене. В этом смысле он является обучающим примером по работе с дисплеем в ActionScript 3.0.

Второй пример будет проигрывать *mp3*-файл. В большинстве случаев, когда вы хотите что-то проиграть, будь это *mp3*-файл или видео, вы хотите слышать и видеть только один из них за раз. Проигрывание одновременно Баха и песни Гнарлса Баркли может создать такую какофонию, которую ни Иоганн Себастьян, ни Гнарлс не захотели бы никогда услышать. Класс типа Одиночка поможет воспроизводить их мелодии по очереди, по одной за раз.

В третьем примере будет рассмотрена обычная тележка для покупок в интернет-магазине для иллюстрации того, как одиночный представитель может быть использован для подсчета общей суммы покупок. В ситуациях, когда вашему приложению требуется абсолютно, «безвопросов», единственный представитель для ведения финансовых подсчетов, использование шаблона Одиночка может быть просто спасением. В этом примере будет также показано, как связать класс с созданным на сцене клипом и встроенным текстовым полем.

Одиночное важное сообщение

Поскольку ясное и недвусмысленное оповещение пользователя является важной задачей для любого хорошего сайта, смешанные сообщения должны всячески исключаться из любых его веб-приложений. Используя шаблон Одиночка, вы можете гарантировать, что у вас есть только один источник всех сообщений для пользователя. При этом резко снижается вероятность того, что вы пошлете ему два противоречащих друг другу сообщения. Следующее приложение использует шаблон Одиночка для указанной цели. Оно разработано так, что где бы ни возникло новое сообщение для пользователя, существует только один элемент для его доставки.

Для начала откройте новый файл ActionScript и введите в него сценарий из нижеследующего примера 3.10.

Пример 3.10. Alert.as

```
package
{
    public class Alert
    {
        private var _msg:String;
        private static var _instance:Alert;
        public function Alert(pvt:PrivateClass) {

        }
        public static function getInstance():Alert
        {
            if(Alert._instance == null)
            {
                Alert._instance=new Alert(new PrivateClass());
                trace("Alert instantiated");
            }
            return Alert._instance;
        }
        // свойства класса
         public var colorFont:uint;
        // методы класса
        public function getMsg():String
        {
            return _msg;
        }
        public function setMsg(alert:String):void
        {
            _msg = alert;
        }
    }
}

class PrivateClass
```

```
{
    public function PrivateClass() {
    trace("PrivateClass called");
    }
}
```

Сохраните этот файл под именем *Alert.as*. Как видите, совсем не обязательно сохранять шаблон проектирования Одиночка в файле под именем «*Singleton*»; единственной причиной использования такого имени в предыдущих примерах было наше стремление помочь вам разобраться в его структуре и месте, которое он занимает в приложении. На самом деле наш файл практически идентичен файлам *Singleton.as*, которые мы уже использовали в примерах этой главы, и даже имеет те же методы, что и класс из примера 3.8. Однако данный пример включает в себя свойство, используемое для задания цвета шрифта для используемого здесь текстового поля. Вполне возможно, что вашему приложению могут потребоваться сообщения разного цвета для отображения различных их категорий.

В ActionScript 3.0 вы можете создать представителя класса `TextField` («текстовое поле») одним из двух способов. Вы можете создать его динамически в своем сценарии, как это будет реализовано в данном примере. Или вы можете создать объект класса `MovieClip` на сцене, включить в него `TextField`, а потом создать ссылку на это встроенное текстовое поле. Но если у вас нет серьезных оснований так делать, более простым на практике и менее затратным по памяти является простое использование `TextField` в вашем сценарии. (Позднее в этой главе клип в форме тележки для покупок со встроенным текстовым полем покажет нам, как использовать встроенный объект `TextField` на сцене.)

Откройте новый файл ActionScript, введите в него код из примера 3.11, назовите этот файл *AlertTest.as* и сохраните его в том же каталоге, где содержится файл *Alert.as*.

Пример 3.11. AlertText.as

```
package
{
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldAutoSize;
    import flash.display.Sprite;
    public class AlertTest extends Sprite
    {
        public function AlertTest()
        {
            var reminder:Alert = Alert.getInstance();
            reminder.setMsg("Remember to use addChild()");
            var announce:TextField=new TextField();
            announce.text=reminder.getMsg();
            announce.autoSize=TextFieldAutoSize.LEFT;
```

```
            var format:TextFormat=new TextFormat();
            reminder.colorFont=0x990099;
            format.color=reminder.colorFont;
            format.font="Arial Black";
            announce.setTextFormat(format);
            announce.x=200;
            announce.y=150;
            this.addChild(announce);
        }
    }
}
```

Когда вы используете что-то менее стандартное, чем команда `trace()`, в вашем приложении, вам надо импортировать в него все необходимые пакеты и классы. Кроме того, все, что вы планируете разместить на сцене, должно быть либо классом `Sprite`, либо `MovieClip`. Поскольку в нашем приложении не используется временная диаграмма, мы можем использовать класс `Sprite`. Таким образом, класс `AlertTest` расширяет класс `Sprite`.

Нам хотелось бы упомянуть здесь пару концепций для новичков в ООП и тех, у кого мало опыта в использовании ActionScript 3.0, – *экономию пакетов* и *абстрактность*. Как уже отмечалось в главе 1, абстрактность – один из столпов ООП, и вы можете видеть ее использование в классе `Alert`, в котором параметры и возвращаемые значения являются абстрактными переменными. Концепция *экономии пакетов* напоминает нам о том, что при импорте пакета из базового набора вы должны импортировать только то, что вам действительно нужно, а не весь этот пакет. Как вы можете видеть, для класса `AlertTest` импортируются только необходимые классы. Хотя было бы очень соблазнительно использовать здесь групповой символ звездочка (*), чтобы забрать себе весь их пакет и не печатать несколько конкретных имен классов, тогда мы получили бы себе много дополнительного и ненужного багажа. Например, вместо набора трех строчек

```
import flash.text.TextField;
import flash.text.TextFormat;
import flash.text.TextFieldAutoSize;
```

мы могли бы просто напечатать

```
import flash.text.*;
```

Но при этом мы добавили бы себе 15 дополнительных, лишних классов. То есть экономия нескольких строчек обременила бы наше приложение всеми этими неиспользуемыми классами.[1]

[1] На самом деле при компиляции Flash отсеивает все неиспользуемые классы, даже если они были указаны как импортируемые, и в *swf*-файл они не попадают. Однако это увеличивает время компиляции приложения, и потому считается плохим тоном. – *Прим. науч. ред.*

Вам может показаться довольно странным использование абстрактности для единственного свойства `colorFont`, включенного в класс-одиночку `Alert`. Свойство `colorFont` является абстракцией целого числа без знака (`uint`). Как только мы присваиваем этому свойству конкретное значение, мы тут же присваиваем его свойству `format.color`. Очевидно, что мы могли сразу присвоить это конкретное значение свойству `format.color`, но смысл хранения свойства `colorFont` в Одиночке состоит в обеспечении того, что будет только один источник информации о цвете нашего сообщения. Таким образом, придание подобной абстрактности обычному целому числу без знака в классе-одиночке `Alert` гарантирует нам то, что не возникнет никакой путаницы с цветом.

Чтобы закончить данный пример, откройте новый документ **Flash и введите** `AlertTest` в поле класс документа. Сохраните *fla*-файл в том же каталоге, где находятся файлы *Alert.as* и *AlertTest.as*. На рис. 3.6 показано то, что вы должны увидеть на сцене и в окне вывода.

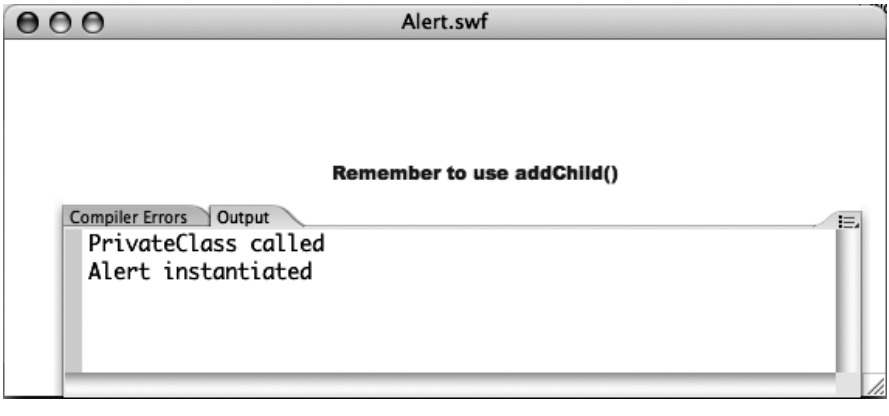


Рис. 3.6. *Выведенное на сцену сообщение*

В более традиционном варианте подобного приложения вы бы использовали не только динамический ввод самого сообщения, но и изменение его цвета. Например, вы могли бы создать систему обратной связи для пользователя, использующую различные цвета и сообщения в зависимости от действий пользователя. Однако в данном примере мы рассмотрели только основы такой системы, с использованием Одиночки и текстового поля.

Только одна мелодия за раз

С наступлением эпохи *mp3*-файлов и *mp3*-плееров музыка перекочевала с CD-дисков в *mp3*-плееры и на устройства, подобные iPod фирмы Apple. Представьте себе, что вы включаете ваш iPod, чтобы послушать любимые мелодии, но все они начинают проигрываться не по очереди, а одновременно. Программное обеспечение, которое работает в вашем

iPod, не позволит такой неразберихе произойти, и вы тоже не должны позволять такому случиться в ваших приложениях. Это еще одна ситуация, где шаблон Одиночка может оказаться очень полезным.

Для начала нам нужно рассмотреть, как мы вообще можем проигрывать *mp3*-файлы с помощью Flash CS3. Мы даже не будем рассматривать идею импортирования *mp3*-файла в главную временную диаграмму (Timeline), которое привело бы к непомерному раздуванию нашего *swf*-файла. Нам, скорее, нужно понять, как ActionScript 3.0 работает со звуками и внешними *mp3*-файлами.

Классы, имеющие дело со звуками, находятся в пакете `flash.media`. Из пяти звуковых классов нам потребуются только классы `Sound` и `SoundChannel`. Для доступа к внешним *mp3*-файлам нам еще понадобится класс `flash.net.URLRequest`. Минимальная последовательность действий для выполнения нашей задачи показана в следующем фрагменте кода:

```
var mySound:Sound = new Sound();
var myChannel:SoundChannel = new SoundChannel;
var myTune:URLRequest= new URLRequest(string);
mySound.load(myTune);
myChannel=mySound.play();
```

Используя следующий скрипт, мы можем создать законченный метод:

```
public function playMe(song:String):void
{
    _tuneUp=new Sound();
    _tuneIn=new URLRequest(song);
    _tuneUp.load(_tuneIn);
    _goChannel=_tuneUp.play();
}
```

Данная функция имеет единственный параметр – строковую переменную, которая будет именем песни (а также именем ее *mp3*-файла). Вы можете думать об этом параметре как об URL-адресе *mp3*-файла песни, поскольку им он, по сути, и является. Однако имеет больше смысла обозначать им отдельную песню, ведь в каждом *mp3*-файле у нас будет храниться только одна песня или мелодия. Так как данный метод будет находиться в классе шаблона Одиночка, который будет существовать в единственном экземпляре, это гарантирует, что каждый раз будет проигрываться только одна мелодия.

Но прежде чем пойти дальше, нам нужно рассмотреть, как работает класс `SoundChannel`. Как только будет задействован метод `Sound.play()` представителя этого класса, начнется проигрывание песни. Если, используя тех же представителей классов `Sound` и `SoundChannel`, вы загрузите и запустите проигрывание другого *mp3*-файла, то его проигрывание также начнется. Согласно логике представителей этих классов у них нет никаких других обязательств, поскольку они уже запустили процесс с первым *mp3*-файлом и, таким образом, закончили свою работу с ним. Следовательно, они готовы повторить это и с другим *mp3*-файлом, даже

если первый еще не закончил проигрываться. Две мелодии наложатся друг на друга, и, поскольку именно подобного поведения мы и хотим избежать, наш код должен проверять, не имеется ли поблизости еще один представитель класса SoundChannel с присвоенным ему значением. Это делается сравнением значения такого представителя с константой null. Если такое равенство обнаруживается, скрипт может спокойно продолжить свою работу дальше и активировать воспроизведение выбранной мелодии. Но если встречается значение не null, то лучше остановить проигрывание любых звуков, ассоциированных с текущим представителем SoundChannel, до загрузки и проигрывания нового *mp3*-файла. Таким образом, вы сможете быть уверены, что у вас не будет проигрываться несколько мелодий одновременно. Мораль для этого приложения состоит в следующем:

> Простое использование шаблона Одиночка еще не означает, что в вашем приложении будет единственный результат его работы. Вы должны таким образом спроектировать вашего Одиночку, чтобы гарантированно получать от него единственный результат так, как вам это нужно.

Имея в виду вышесказанное, мы можем теперь создать нашего Одиночку, гарантирующего, что только одному *mp3*-файлу разрешено проигрываться в любой момент времени. Введите скрипт, приведенный в примере 3.12, в файл ActionScript и сохраните его в файле *Tuner.as*.

Пример 3.12. Tuner.as

```
package
{
    import flash.net.URLRequest;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    public class Tuner
    {
        private var _goChannel:SoundChannel;
        private var _tuneUp:Sound;
        private var _tuneIn:URLRequest;
        // реализация Одиночки
        private static  var _instance:Tuner;

        // конструктор Одиночки
        public function Tuner (pvt:PrivateClass)
        {
        }
        // метод, создающий Одиночку
        public static  function getInstance ():Tuner
        {
            if (Tuner._instance == null)
            {
                Tuner._instance=new Tuner(new PrivateClass  );
```

```
                trace (“Tuner instantiated”);
            }
            return Tuner._instance;
        }
        // Начать проигрывание
        public function playMe (song:String):void
        {
            if (_goChannel != null)
            {
                _goChannel.stop ();
            }
            _tuneUp=new Sound  ;
            _tuneIn=new URLRequest(song);
            _tuneUp.load (_tuneIn);
            _goChannel=_tuneUp.play();
        }
        // Остановить проигрывание
        public function stopMe ():void
        {
            if (_goChannel != null)
            {
                _goChannel.stop ();
            }
        }
    }
}

class PrivateClass
{
    public function PrivateClass ()
    {
        trace (“PrivateClass called”);
    }
}
```

Обратите внимание, что данный скрипт реализации Одиночки сам импортирует пакеты и классы. До этого момента весь импорт происходил в скриптах, которые вызывали класс-одиночку. Однако какой бы метод или свойство, включая абстрактные, не использовали ссылки на внешние классы и пакеты, все они должны быть где-то импортированы. Вы увидите, например, что класс (`DoMusic`), который будет имплементировать класс-одиночку (`Tuner`), теперь не будет нуждаться в каком-либо импорте объектов из других пакетов, поскольку все они уже импортированы в классе Одиночки.

Так как данное приложение всего лишь гарантирует, что только один *mp3*-файл будет проигрываться в каждый момент времени, то имплементация Одиночки будет в нем очень простой. Пример 3.13 демонстрирует код для данного класса:

Пример 3.13. DoMusic.as

```
package
{
    import flash.display.Sprite;
    public class DoMusic extends Sprite {
        public function DoMusic() {
            var playOne:Tuner = Tuner.getInstance();
            // Используйте любой доступный вам mp3-файл
            playOne.playMe("blues.mp3");
        }
    }
}
```

Сохраните пример 3.13 в файле *DoMusic.as* в том же каталоге, где содержится файл *Tuner.as*. Откройте файл документа **Flash, в поле его класса** на панели свойств введите `DoMusic` и сохраните его под именем *Music.fla* в каталоге, где содержатся наши два файла ActionScript. Наконец, поместите любой *mp3*-файл в тот же каталог и переименуйте его в *blues.mp3*. (Вы можете оставить его прежнее имя, но тогда измените название в ссылке на него из скрипта с *blues.mp3* на имя вашего *mp3*-файла; главное, чтобы имя файла и название в ссылке были одинаковыми.) Когда вы запустите ваше приложение, вы должны будете услышать, как проигрывается ваш *mp3*-файл.

В более реалистичном примере подобное приложение позволяет нам выбирать мелодию для воспроизведения из нескольких возможных вариантов, а Одиночка гарантирует, что только одна из них будет проигрываться за раз. Метод для проигрывания *mp3*-файлов принадлежит классу Одиночка, поэтому если будет выбрана другая мелодия, то будет задействован единственный представитель этого класса для ее воспроизведения. Таким образом, данный Одиночка должен совершить всю работу со звуками и убедиться, что любой представитель класса `SoundChannel`, проигрывающий музыку, будет остановлен; то есть он делает гораздо больше, чем просто заботится о единственности своего представителя. В примере 3.14 находится весь необходимый для такой реализации код. Сохраните его в файле *DoMusicBtn.as* в том же каталоге, где содержится файл *Tuner.as*.

Пример 3.14. DoMusicBtn.as

```
package
{
    import flash.events.MouseEvent;
    import flash.display.Sprite;

    public class DoMusicBtn extends Sprite
    {
        private var _playOne:SongPlay=new SongPlay();
        private var _playTwo:SongPlay=new SongPlay();
```

```
        private var _stopSong:SongPlay=new SongPlay();
        private var playTune:Tuner;

        public function DoMusicBtn ()
        {
            playTune = Tuner.getInstance();
            // Ставим кнопки
            addChild (_playOne);
            _playOne.x=30;
            _playOne.y=50;
            addChild (_playTwo);
            _playTwo.x=30;
            _playTwo.y=150;
            addChild (_stopSong);
            _stopSong.x=130;
            _stopSong.y=100;

            // Подключаем обработчики событий
            _playOne.addEventListener (MouseEvent.CLICK,doOne);
            _playTwo.addEventListener (MouseEvent.CLICK,doTwo);
            _stopSong.addEventListener (MouseEvent.CLICK,doStop);
        }
        // Методы
        function doOne (e:MouseEvent):void
        {
            playTune.playMe ("blues.mp3");
        }
        function doTwo (e:MouseEvent):void
        {
            playTune.playMe ("class1.mp3");
        }
        function doStop (e:MouseEvent):void
        {
            playTune.stopMe ();
        }
    }
}
```

Чтобы протестировать данную конкретную имплементацию, вам потребуется немного более сложный файл документа Flash, чем обычно. Следующие шаги показывают, что вы должны сделать.

1. Откройте новый документ Flash и сохраните его в том же каталоге, где содержится ваш файл *Tuner.as*.
2. Используя инструмент рисования «овал», нарисуйте на сцене овал радиусом 15. Выделите его, а потом нажмите F8, чтобы открыть диалоговое окно конвертации в символ. Введите SongPlay в поле Name (Имя) и в качестве типа выберите Button (Кнопка). Кликните на кнопке выбора Export for ActionScript (Экспорт для ActionScript) в группе Linkage (Связь). В имени класса должно появиться SongPlay, а в имени базово-

го класса – flash.display.SimpleButton. Кликните на OK. Удалите кнопку со сцены. (Она останется в Library (Библиотеке).)

3. Откройте панель свойств и в поле класса документа введите DoMusicBtn. Сохраните файл еще раз.
4. В каталог, в которой вы сохранили файлы ActionScript и документ Flash, добавьте два *mp3*-файла, используя имена из примера 3.14. (Вы можете изменить либо имена ваших *mp3*-файлов, либо имена в примере 3.14.)
5. Протестируйте ваше приложение, нажав Ctrl + Enter (Command + Return для Макинтош).

Вы должны увидеть появление трех кнопок. С помощью двух кнопок слева вы можете выбирать один из двух *mp3*-файлов, а одна кнопка справа должна останавливать воспроизведение файла, какой бы он ни был. Если вы кликните на любой кнопке слева во время проигрывания мелодии, то оно будет остановлено и выбранная мелодия начнет воспроизводиться с самого начала.

Использование множественных ссылок в тележке для покупок

Этот последний пример использования шаблона проектирования Одиночка показывает, что на представителя его класса может быть несколько ссылок, и вместе с тем в его работе сохраняются порядок и аккуратность. Как и все предыдущие примеры, данный сделан довольно простым, чтобы мы могли сосредоточить свое внимание на роли, которую играет в нем шаблон Одиночка, и на некоторых деталях его структуры.

Интернет-тележка для покупок – это на самом деле не более чем образное представление простого цифрового накопителя. Как только в нее добавляется новый товар, его цена аккумулируется; и когда приходит время расплачиваться, можно легко рассчитать полную стоимость покупок, налог с продаж и стоимость доставки. Такая тележка является хорошим примером применения шаблона Одиночка, поскольку вы должны быть уверены, что у вас только одна тележка, неважно, где вы добавляете в нее товары или удаляете их оттуда.

Для начала пример 3.15 показывает нам класс-одиночку с единственным методом добавления к текущей общей стоимости. Откройте новый каталог, скопируйте код из примера 3.15 и сохраните его в файле *ShopCart.as*.

Пример 3.15. ShopCart.as

```
package
{
    public class ShopCart
    {
        private static var _cartTotal:Number=0;
```

```
        private static var _instance:ShopCart;
        public function ShopCart(secure:PrivateClass) {
            trace("ShopCart instantiated");
        }
        public static function getInstance():ShopCart
        {
            if(ShopCart._instance == null)
            {
                var security:PrivateClass=new PrivateClass();
                ShopCart._instance=new ShopCart(security);
            }
            return ShopCart._instance;
        }
        // Метод
        public function kaChing(cart:Number):Number
        {
            _cartTotal+=cart;
            return _cartTotal;
        }
    }
}

class PrivateClass
{
    public function PrivateClass() {
        trace("private class installed");
    }

}
```

Как и в предыдущих примерах, здесь в методе используется приватная статическая переменная `_crtTotal`. При вызове метода `kaChing()` она инкрементируется с помощью составного оператора (+=) и затем возвращает текущую общую сумму.

Чтобы показать значение шаблона Одиночка в данном приложении, скрипт из примера 3.16 создает его представителя с двумя разными именами. Первая ссылка называется `myCart` («моя тележка»). С помощью метода `kaChing()` в нее добавляются три продукта, и результат помещается в текстовое поле `MovieClip`. Затем вторая тележка под именем `ghostCart` («неизвестная тележка») в окне Output пытается что-то добавить к сумме, уже показанной у нашей тележки. Как вы увидите, вторая ссылка использует того же представителя Одиночки, поскольку происходит просто добавление значения `ghostCart` к значению графической тележки для покупок, что и выводится в окне Output. Это демонстрирует нам единственность нашего представителя и гарантирует, что даже второе имя ссылки на тележку для покупок не помешает нам корректно вычислить общую сумму. Введите скрипт из примера 3.16 и сохраните его как *TestCart.as* в каталоге с файлом *ShopCart.as*.

Пример 3.16. TestCart.as

```
package
{
    import flash.display.Sprite;
    public class TestCart extends Sprite
    {
        private var cartNow:Cart=new Cart();
        public function TestCart ()
        {
            var myCart:ShopCart = ShopCart.getInstance();
            var apples:Number=2.45;
            var oranges:Number=3.50;
            var blackberries:Number=4.11;
            myCart.kaChing (apples);
            myCart.kaChing (oranges);
            myCart.kaChing (blackberries);
            var total:String=String(myCart.kaChing(0));
            cartNow.total_txt.text="$"+total;
            this.addChild (cartNow);
            cartNow.x=(550/2)-(321.6/2);
            cartNow.y=(400/2)-(316.6/2);
            // Попытка создать нового представителя и записать новое число
            var ghostCart:ShopCart = ShopCart.getInstance();
            trace ("Ghost cart adds 25.33 for a total of $"
                    +ghostCart.kaChing(25.33));
        }
    }
}
```

Последним этапом является создание на сцене клипа с текстовым полем, который будет представлять нашу тележку для покупок. Хотя из кода довольно легко создать и вывести на сцену текстовое поле, из него не так просто нарисовать тележку для покупок. Конечно, для такой задачи существует решение во **Flash. Там вы можете создавать изображения**, а потом контролировать их из вашего кода. Однако все это потребует гораздо большего количества действий, чем простое введение имени класса документа Flash. Поэтому используйте следующие шаги для создания модуля тележки для покупок:

1. Откройте новый документ Flash и сохраните его как *TestCart.fla* в том же каталоге, где содержится ваш файл *ShopCart.as*.
2. С помощью инструментов рисования нарисуйте тележку для покупок. Используя статический текст, поместите в ее центре надпись «Total» (Сумма). Выделите всю созданную тележку и нажмите F8, чтобы открыть диалоговое окно конвертации в символ. Введите Cart в поле Name и выберите Movie clip в качестве ее типа. Пометьте кнопку выбора Export for ActionScript и нажмите на OK. (Рисунок 3.7 показывает пример подобной тележки.)

3. Щелкните два раза на клипе тележки для покупок, чтобы открыть ее для редактирования. Используя инструмент Text (Текст), добавьте к ней динамическое текстовое поле. Дайте ему имя `total_txt` на панели свойств.
4. Выйдите из режима редактирования символа, кликнув на иконке Scene 1 (Сцена 1). Удалите клип со сцены. (Не волнуйтесь, он останется на панели Library.)
5. Протестируйте ваше приложение, нажав Ctrl + Enter (Command + Return для Макинтош).

В ходе своей работы это приложение создает единственного представителя класса `ShopCart` и, используя метод `kaChing()`, добавляет товаров на общую сумму $10.06 в текстовое поле клипа тележки для покупок, как это показано на рис. 3.7.

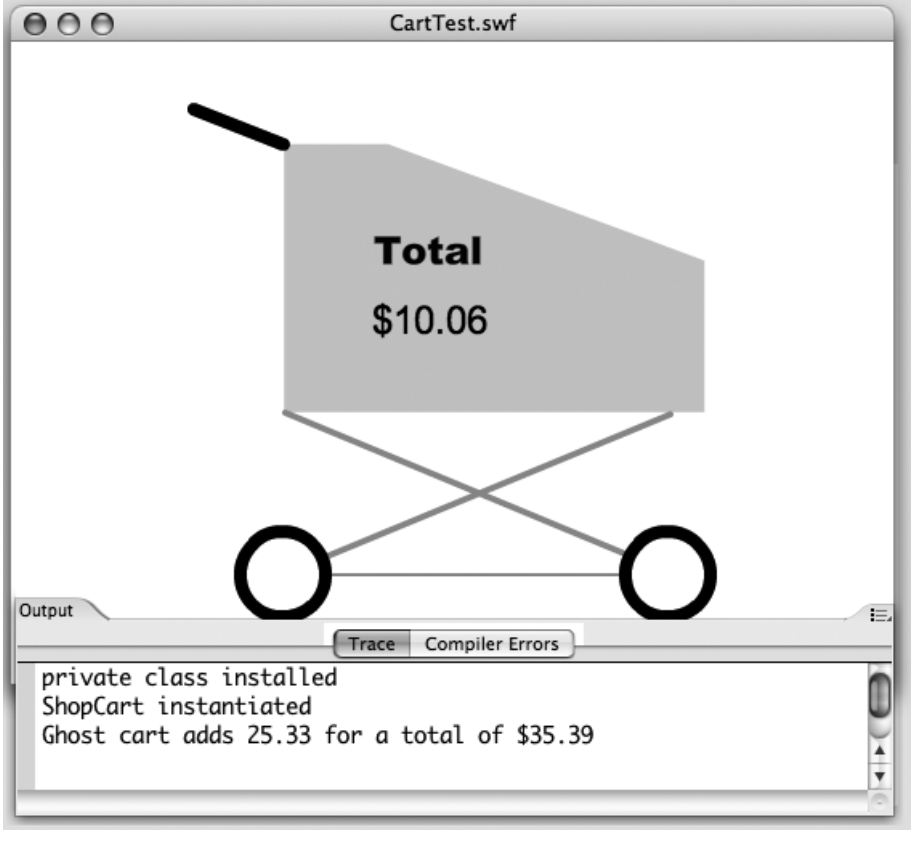


Рис. 3.7. *Корректный результат с двумя ссылками на одного представителя*

На рис. 3.7 показано также окно Output, в котором в тележку ghostCart добавляется товар на сумму $25.33, что дает в итоге $35.39. Это просто сумма 10.06 и 25.33. Значение $10.06 составляет текущую итоговую сумму ссылки myCart. Если к ней добавить значение из ссылки ghostCart, мы получим 35.39, как и было предсказано.

Заключение

Шаблон Одиночка гарантирует нам, что только один представитель его класса будет создан в приложении и к нему будет обеспечен глобальный доступ. Хотя некоторые аспекты Одиночки кажутся простыми, прочие его возможности показывают, насколько данный шаблон проектирования может быть искусным и сложным. Конечно, можно придумать много способов создания классов-одиночек, и авторы настоящей главы были в этом вовсе не оригинальны. Однако они представили очень стандартную реализацию Одиночки, а также одну такую реализацию, которая решает некоторые специфические проблемы ActionScript и, вообще говоря, ECMAScript.

Шаблон Одиночка, особенно его классическая схема реализации, на основе которой мы построили всю эту главу, имеет множество хорошо аргументированных претензий к себе. Даже короткий поиск по посвященным шаблонам проектирования книгам и статьям, включая замечательные экземпляры, доступные в Интернет, даст вам множество различных идей, как улучшить схему класса Одиночки или даже как совсем заменить его другими концептуальными структурами. Тот факт, что Одиночка провоцирует такие мысли, является следствием его свойств, и нам самим, как вы помните, приходилось бороться с его ограничениями. Так что, если и вы придумаете для него лучшую схему или альтернативу, без сомнений воспользуйтесь ею.

Наиболее частой претензией к Одиночке является его некорректное использование, или использование в контексте, где он начинает мешать гибкости приложения или повторному использованию его кода. Другая претензия состоит в том, что шаблон применяют слишком уж часто. Частично это можно отнести к его простоте, но иногда злоупотребление им происходит из-за неполного понимания разработчиками целей и задач данного шаблона. Одна из причин, почему мы рассмотрели здесь так много разных примеров использования Одиночки, состоит в том, что мы хотели показать, что он имеет свои явные области применения. Запомните: не нужно использовать шаблон Одиночку там, где велика угроза изменений или необходимость в повторном использовании кода. У Одиночки очень четкое предназначение – ограничить число представителей его класса до одного и обеспечить к нему глобальную точку доступа. Вне этих пределов он весьма ограниченно полезен, и хотя он является популярной частью других шаблонов проектирования, вы

должны быть с ним осторожны и не позволять ему мешать использованию других шаблонов.

Однако, как мы продемонстрировали в этой главе, у Одиночки есть несколько очень практичных применений, и вы не должны забывать об этом. Как и другие шаблоны проектирования, Одиночка призван решать определенный круг программистских задач. Тот факт, что вы не можете применять его всякий раз при их возникновении, вовсе не означает, что шаблон Одиночка устарел. Скорее, это будет означать, что Одиночка не является лучшим решением для вашей конкретной ситуации. В случае таких задач, особенно повторяющихся, следует либо адаптировать под них шаблон Одиночка, либо использовать другой шаблон, либо разрабатывать и регулировать определенные аспекты классической схемы реализации этого шаблона.

III

Структурные шаблоны

Истинное украшение не является результатом улучшения внешнего окружения. Оно органично связано со структурой того, что оно украшает, будь это человек, здание или парк.

Франк Ллойд Райт

Важность некоторых проблем, касающихся фактов, будет встроена в структуру системы.

Талкот Парсонс (социолог XX века)

Если структура и компоненты программы были выбраны хорошо, то часто многие из их внутренних инструкций могут быть сохранены без изменений.

Никлаус Вирт

Структурные шаблоны – это некоторые стратегии, использующие классы и объекты для построения более сложных структур. Частично процесс вызывает применение абстрактных классов как интерфейсов, производные классы от которых составляют более крупные структуры. Используя наследование и композицию, структурные шаблоны обеспечивают большую гибкость для разработчиков приложений. Шаблоны, работающие с классами, имеют тенденцию использовать наследование для порождения больших структур, а шаблоны, работающие с объектами, – композицию. Однако оба эти подхода имеют одну общую цель. Рисунок III.1 иллюстрирует в общем виде структурные шаблоны.

Мысленно вы можете представлять себе конструирование приложений с помощью структурных шаблонов как сборку из деталей, вынимаемых из разных лотков. Лотки символизируют различные расширения клас-

сов и композиции объектов. Хотя новая структура порождается из деталей лотков, она становится новым явлением, отличающимся от простой суммы своих исходных частей.

В данной части рассматриваются три различных структурных шаблона. Первый из них, Декоратор, добавляет объекту обязанностей. Он как бы *упаковывает* его в свою новую функциональность и свойства, в то время как внутренняя структура объекта остается по сути неизменной. Шаблон Адаптер обеспечивает возможность двум не связанным между собой интерфейсам работать совместно. При добавлении объекта к несовместимой с ним напрямую структуре Адаптер дает альтернативу полному переписыванию приложения заново для добавления в него новых возможностей. Вы узнаете, как создавать Адаптеры с помощью *наследования* и *композиции*.

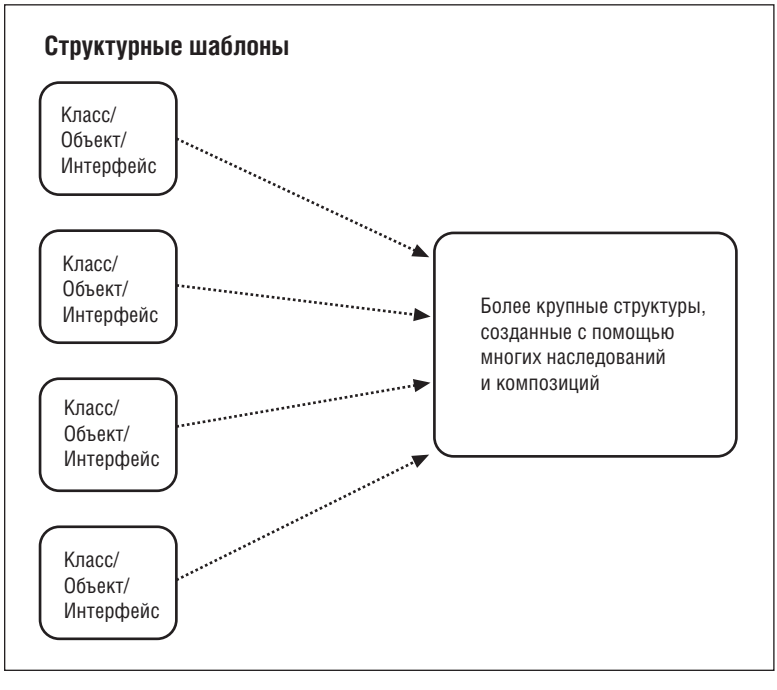


Рис. III.1. *Структурные шаблоны проектирования*

И наконец, шаблон Композиция собирает объекты в древовидные структуры так, что клиенты могут рассматривать простые и составные объекты единым образом. Структурируя объекты подобным способом, шаблон Композиция сильно упрощает создание комплексных структур.

Глава 4 «Шаблон Декоратор»

Глава 5 «Шаблон Адаптер»

Глава 6 «Шаблон Композиция»

4

Шаблон Декоратор

Я пытаюсь приукрасить
свое воображение как только могу.

Франц Шуберт

Картины чаще загораживают
собой стены, чем украшают их.

Уильям Вордсворт
(поэт и человек, который действительно
умел тонко чувствовать)

Что такое шаблон Декоратор?

Хотя шаблон Декоратор не разрабатывался специально для разработчиков или проектировщиков веб-приложений, иногда кажется, что это было именно так. Несколько лет назад мы создали веб-сайт, про который знали, что нам придется периодически его обновлять. Он имел такую структуру, что замена одного элемента на новую его версию не составляла труда и не требовала значительных изменений в коде. Поскольку нам приходилось обновлять его не очень часто, это не было большой проблемой. Однако если бы нам было необходимо обновлять элементы сайта на регулярной основе, наш проект и его структура оставляли бы желать много лучшего.

Вообразить себе такую ситуацию, когда требуются частые изменения некоторых частей сайта, не так уж сложно. Например, если ваш заказчик – продавец, регулярно размещающий новые объявления о разнообразных специальных предложениях и товарах, то вам потребуется некоторая гибкость во внутренней структуре вашего сайта, чтобы можно

было быстро реализовывать такие изменения, не переделывая при этом его базовую структуру.

Задачей Декоратора как раз и является поддержание общей структуры с возможностью проведения изменений, осуществляемое при помощи декорирования различных компонентов, составляющих приложение. Декорации состоят из свойств и/или методов, используемых для «упаковывания» в них различных объектов приложения. Как вы увидите, этот шаблон позволяет вам смешивать и сочетать все различные компоненты и декорации для них, чтобы достигнуть оптимальной гибкости и расширяемости приложения при сохранении базовой структуры классов.

Основные характеристики

Достаточно узнать две основные характеристики шаблона Декоратор для его понимания. Часто разработчикам требуется добавить некоторые уникальные возможности какому-то объекту, не добавляя при этом их всему его классу. Как и для других шаблонов проектирования, для шаблона Декоратор характерно добавление новых уникальных возможностей. Но идентифицирующей его характеристикой является то, каким именно уникальным образом он это делает. Его «упаковывание» компонента в новый объект с новыми возможностями следует нескольким принципам:

- Декоратор может применяться в любом месте использования исходного объекта.
- Во время работы программы вы можете смешивать и сочетать комбинации Декораторов так, как это требуется.

Чтобы лучше понять основные характеристики шаблона проектирования Декоратор, нужно рассмотреть несколько других вариантов, делающих то же, что и данный шаблон. Итак, ваш программный проект требует добавления новых возможностей и характеристик отдельному *объекту*, а не всему его *классу*. Использование для этого наследования приведет к непомерному раздуванию класса объекта и к необходимости его изменения с каждой новой добавляемой возможностью; причем каждый новый объект такого класса будет наследовать все его характеристики и функциональность, когда это будет вам совсем не нужно.

В этой главе вы будете часто встречать термин «компонент». Мы будем его использовать совсем не в том смысле, что компоненты во Flash, интерфейсе пользователя, мультимедиа, данных или других областях. В контексте данной главы компоненты будут означать конкретных представителей классов, которые будут декорироваться с помощью других конкретных представителей, называемых *декораторами*. Таким образом, на время изучения этого шаблона не думайте о компоненте иначе, как о чем-то, что декорируется. (В последнем примере приложения данной главы

вы будете использовать компоненты пользовательского интерфейса Flash, но к тому времени вы уже сможете различать эти два вида компонентов.)

Представьте, что вы создаете сайт продавца автомобилей. На нем вы можете выбирать различные модели автомобилей и различные опции для них, такие как MP3-плеер, GPS-навигатор, обивка сидений из ткани, винила или кожи и различные варианты сигнализаций. Если вы будете использовать наследование, то все эти опции будут доступны для каждого объекта. И вам нужно будет хранить также все модели автомобилей в вашем главном классе. Но это же абсурд! Кому могут потребоваться одновременно обивка сидений из ткани и винила или Форд Эскейп вместе с Шевроле Малибу? А если появятся новые опции, вам придется раздувать всеми ими ваш класс и делать их доступными для каждого объекта. Но если бы вы могли просто «упаковать» в новую возможность ваш компонент, когда и если это нужно, вы могли бы сохранить вашу программу стройной, упорядоченной и готовой к изменениям.

Чтобы разобраться в работе шаблона проектирования Декоратор, нужно главным образом понять, что он использует наследование и задействует *абстрактные классы*. Однако, как вы знаете из главы 2, ActionScript 3.0 не поддерживает абстрактные классы. Вы *можете* в нем создавать классы, которые будут работать *подобно* абстрактным классам, просто не создавая напрямую их представителей. Фактически именно этим и является абстрактный класс в некотором приближении – классом, который можно расширять, но представителей которого создавать нельзя. Его работа в чем-то похожа на работу интерфейса, но, хотя вы можете имплементировать интерфейс и все его абстрактные методы, вы не можете его расширять. (Мы немного рассматривали использование интерфейсов в главе 1, а в главе 5 мы будем использовать шаблон проектирования, который задействует интерфейсы.)

Отсутствие поддержки абстрактных классов является больным местом для некоторых разработчиков приложений на Flash и Flex. Но прежде чем писать гневное письмо в компанию Adobe, взгляните на спецификацию ECMASript версии 4. Она тоже в настоящий момент не поддерживает абстрактные классы.[1]

Почему абстрактные классы так важны для шаблона Декоратор? Как и в интерфейсах, вы можете создавать в них абстрактные методы, которые потом могут быть имплементированы различными способами. В то же время вы можете использовать их для создания подклассов, которые будут наследовать их базовую функциональность, что невозможно для интерфейсов.

[1] Слово «abstract» внесено в спецификации ECMAScript в раздел «зарезервированных на будущее», так что, возможно, когда-нибудь поддержка абстрактных классов будет реализована. – *Прим. науч. ред.*

Модель Декоратора

Диаграмма классов шаблона проектирования Декоратор показывает его два ключевых элемента: компонент и декоратор. Компонент описывает собой то, что должно быть декорировано, а декоратор представляет собой абстрактный класс для конкретных декораций. Конкретный компонент – это то, что непосредственно должно быть отдекорировано, а конкретные декорации являются средствами такого декорирования. Все конкретные декорации и конкретный компонент создают подклассы для одного и того же класса. Принимая во внимание вышесказанное, посмотрите на диаграмму классов на рис. 4.1.

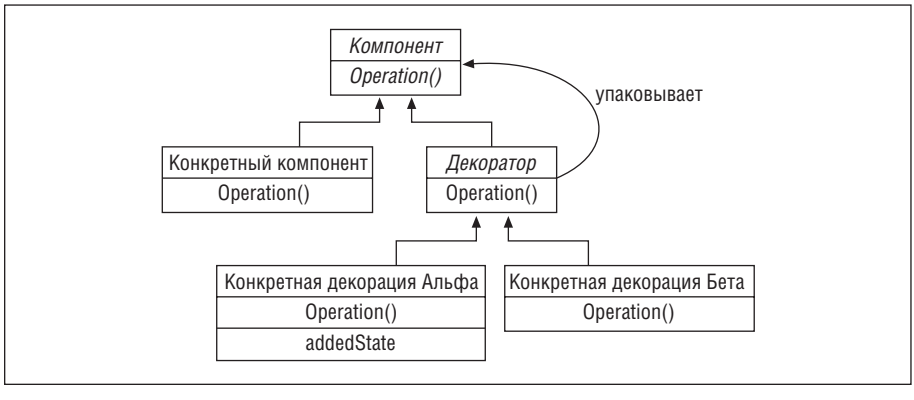


Рис. 4.1. *Шаблон проектирования Декоратор*

В некотором смысле компонент – это рождественская елка, а декоратор – гирлянды на ней. Каждый конкретный декоратор окутывает собой елку и создает подклассы вместе с ней. Еще лучше представлять компоненты и декораторы как вкладывающиеся друг в друга куклы из Японии или России. Самая маленькая кукла является компонентом, он помещается (упаковывается) в декоратор. Затем компонент и декоратор вместе помещаются в другой декоратор, затем в следующий декоратор и так далее. Рисунок 4.2 показывает эту модель.

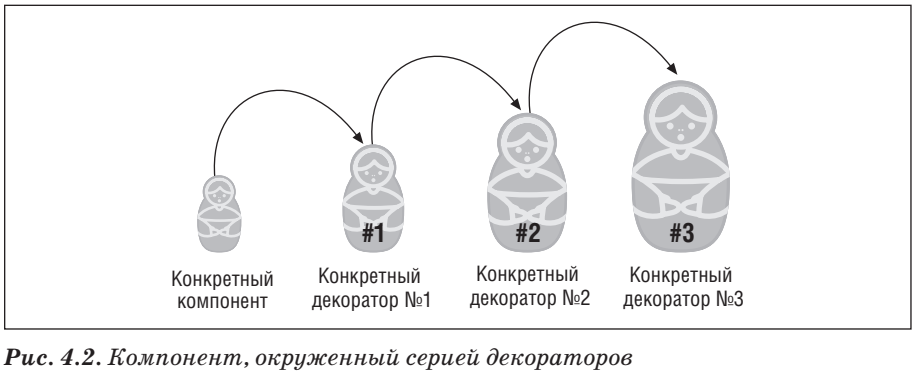


Рис. 4.2. *Компонент, окруженный серией декораторов*

Наверно, самое важное, что нужно запомнить о шаблоне проектирования Декоратор, – это то, что его задачей является добавление дополнительных элементов к классу без изменения базовой структуры класса. Вместо изменения самого класса осуществляется его декорирование с помощью подклассов, которые не пересекаются с их классом-предком.

Ключевые концепции ООП, используемые в шаблоне Декоратор

Иногда, размышляя о такой ключевой концепции ООП как *наследование*, нужно думать и о дальних последствиях ее применения. Полезным свойством наследования является то, что сразу после создания суперкласса все его подклассы наследуют все его характеристики. Однако вам, возможно, и не нужен весь этот набор для каждого подкласса суперкласса. Если вам требуется добавить некоторую функциональность вашему объекту, то создание подклассов может не подойти вам в качестве решения, поскольку все остальное, что вы унаследуете от одного суперкласса, может быть сильно перегружено ненужной функциональностью.

Нежелательное наследование

В конце этой главы вы увидите пример салона по продаже автомобилей, где пользователи выбирают машины с различными вариантами опций. Если бы все опции происходили из одного суперкласса, то они были бы доступны для каждой из машин автосалона независимо от того, хотите вы этого или нет. Одной из опций, например, является камера заднего вида, используемая на минивэнах и больших внедорожниках с ограниченным обзором сзади. Такая опция была бы чрезмерной на таких спортивных машинах, как Шевроле Корветт, Морган или Феррари. Однако при применении подклассов она на них окажется. Каждый подкласс получает от своего суперкласса все его характеристики. Так что это и есть тот случай, когда нам требуется что-то другое вместо простого наследования. Иначе может получиться, что абсолютно все компоненты будут иметь одинаковые характеристики. Мини-грузовик будет подклассом тех же компонентов, что и шикарная машина, и оба унаследуют опции, которые им не нужны и не будут ими использоваться.

Привнесение возможностей

Шаблон проектирования Декоратор известен также под именем Упаковка (или Враппер[1]). Концепция «упаковывания» является сердцем шаблона проектирования Декоратор. Так что же означает «упаковать» один объект в другой? Представьте себе упаковывание подарка – это будет хороший пример принципа работы данного шаблона. Упаковка

[1] От английского «wrapper» – обертка, упаковка. – *Прим. науч. ред.*

несколько трансформирует подарок, но подарок при этом не наследует свойства упаковки, он только использует ее. В отличие от создания подклассов, которое расширяет один класс в другой, упаковка позволяет одному объекту использовать характеристики другого без расширения упакованного объекта или объекта, используемого для упаковки.

Одним из видов преобразования с помощью упаковки, знакомым для большинства программистов на ActionScript 3.0, является трансформация типа данных из одного в другой. Например, класс `Number` может упаковать в себя строковую переменную, после чего она приобретет свойства этого класса. Рисунок 4.3 иллюстрирует описанный традиционный случай упаковки:

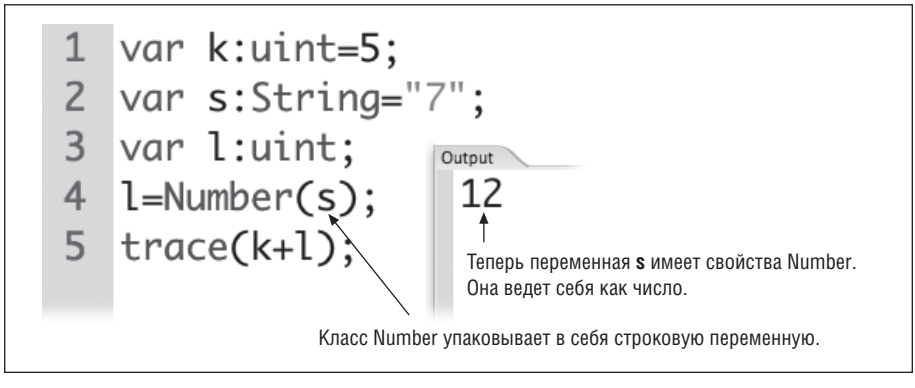


Рис. 4.3. *Упаковка класса String в класс Number*

Иногда мы можем представлять себе упаковывание одного объекта в другой как перехват одним классом обращений программного интерфейса приложения, адресованных определенному представителю другого класса. Пример на рисунке 4.3 показывает, что класс `Number` перехватывает обращение к представителю класса `String`. В результате переменная `l` типа целое без знака может присвоить себе значение `s` как число. Это происходит потому, что переменная `s` упаковывается в класс `Number` и рассматривается как число.

Используя класс декоратор, компоненты упаковываются в свои декорации. Упаковкой являются конкретные представители класса-декоратора, а упаковываются представители класса конкретного компонента. Таким образом, конкретный компонент теперь «содержит в себе» характеристики своей упаковки, но не наследует их. Рисунок 4.4 показывает такой пример.

Конкретный компонент, будучи упакованным в другой класс, приобретает его характеристики. Если эти новые характеристики не нужны, представитель конкретного класса создается просто без упаковки.

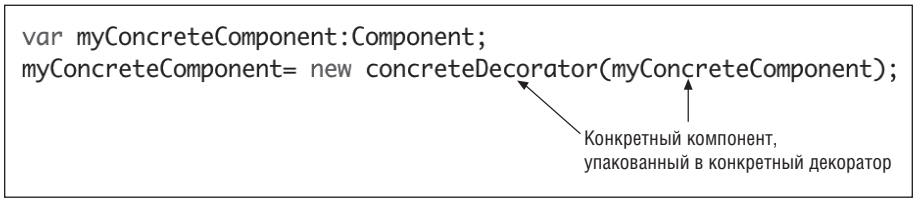


Рис. 4.4. *Конкретный декоратор упаковывает в себя конкретный компонент*

Гибкость и адаптивность

Одна из повсеместно рекомендуемых практик ООП состоит в создании классов, которые могут быть расширены, но не изменены. При изменении класса, особенно имеющего подклассы, вы можете легко разрушить все ваше приложение. Поэтому желательно так создавать ваши классы, чтобы их можно было легко расширить, не изменяя.

Шаблон проектирования Декоратор позволяет вам делать это. Вся его модель основывается на одном классе, являющемся суперклассом для всех остальных его классов. Такой базовый класс представляет собой абстрактный класс компонента для декорирования. Абстрактный класс декоратор является подклассом этого класса и имплементирует заново все его базовые методы. (В контексте использования ActionScript 3.0 *абстрактная* сущность классов симулируется с помощью ключевого слова `override` («заместить»), используемого для ре-имплементации методов в подклассах.) Все другие конкретные классы компонентов и декораторов имеют один корневой суперкласс, даже хотя классы декораторов происходят от абстрактного класса декоратора, а классы компонента – от абстрактного класса компонента. Поэтому, когда объекты конкретного компонента упаковываются в объекты конкретного декоратора, оба этих объекта имеют один и тот же корневой суперкласс. Таким образом, декорированные объекты могут сохранять базовый объект без модификаций, меняя его возможности.

В рамках ООП шаблон Декоратор может добавлять и убирать часть функциональности объекта, упаковывая отобранные компоненты в отдельные декораторы. Кроме того, он может добавлять функциональность объекту более одного раза, упаковывая один из декораторов в другой. В результате этот шаблон проектирования может многократно менять функциональность, не меняя структуру.

Минималистский абстрактный Декоратор

Чтобы начать изучение шаблона проектирования Декоратор, нам потребуется как минимум имплементировать следующие классы:

- Абстрактный компонент
- Абстрактный декоратор

- Конкретный компонент
- Конкретные декораторы

У нас будет два декоратора, чтобы вы могли лучше понять, как работает такая структура (и это будет ближе к модели, показанной на рис. 4.1). Самый важный из всех классов, разрабатываемых в данном примере, – исходный абстрактный класс, моделирующий компонент. Все остальные классы являются подклассами этого базового класса.

Абстрактный класс компонента

Чтобы начать наш процесс реализации шаблона Декоратор, создадим его базовый компонент. Вы можете представлять себе компонент как ненаряженную новогоднюю елку или даже человека, решающего, что ему надеть. Создать базовый компонент просто, потому что все добавляемые характеристики привносятся декорациями. Пример 4.1 имеет только одну строковую переменную, которая будет использоваться для описания нашего компонента. Сохраните этот скрипт в файле *Component.as*.

Пример 4.1. Component.as

```
package
{
    // Абстрактный компонент шаблона Декоратор
    //**************
    // Абстрактный класс
    public class Component
    {
        internal var information:String;

        public function getInformation():String
        {
            return information;
        }
    }
}
```

Начиная с данного момента, все другие классы шаблона Декоратор будут расширять класс `Component`. Помните, что это абстрактный класс (или, по крайней мере, он должен рассматриваться как таковой), и его основной функцией является создание базовой структуры для остального нашего приложения. Он состоит из одной переменной `information` и одной функции-получателя `getinformation()`. Эти элементы будут присутствовать в конкретных компонентах и в декораторах. Компоненты и декораторы должны выдавать нам информацию о своих характеристиках. Конкретная новогодняя елка показывает нам информацию о себе, позволяющую нам отличить ее от других поддающихся декорированию объектов, таких как дворик перед домом, который можно украсить с помощью гномов и пластиковых розовых фламинго. Кроме

того, вы захотите сами запрашивать информацию, поэтому она имеет функцию-получатель.

Абстрактный класс Декоратора

Абстрактный декоратор является подклассом класса `Component` и наследует от него переменную `information`, что нам и нужно. На самом деле для данного примера нам ничего не требуется, кроме определения данного класса как расширения класса `Component`. Однако метод `getinformation()` нужно переопределить независимо от класса `Component`, используя выражение `override`, которое позволяет заместить метод родительского класса. Это делается для того, чтобы была возможность отличать такой же метод, используемый в классе `Decorator`, от метода, используемого в классе `Component`. Все конкретные декораторы происходят от класса `Decorator`, а все конкретные компоненты – от класса `Component`. В дальнейшем конкретные компоненты будут упакованы в конкретные декораторы, и такое разделение становится весьма важным в более сложных имплементациях шаблона проектирования Декоратор. Выражение `trace()` используется, чтобы показать вам, в каком процессе задействован абстрактный класс `Decorator`. Сохраните пример 4.2 в файле *Decorator.as*.

Пример 4.2. Decorator.as

```
package
{
    // Абстрактный декоратор в шаблоне Декоратор
    //**************
    // Абстрактный класс
    public class Decorator extends Component
    {
        trace("|*|Decorator|*|");
        override public function getInformation():String
        {
            return information;
        }
    }
}
```

Итак, два абстрактных класса `Component` и `Decorator` созданы, и пора приниматься за конкретные классы. Для данного примера будет создан только один конкретный компонент. Разумно названный `Concrete-Component`, этот класс представляет все, что будет впоследствии как-то декорировано в шаблоне проектирования Декоратор. У вас может быть множество конкретных компонентов, использующих один и тот же набор декораторов, или только один. Позднее в данной главе вы увидите пример приложения, в котором несколько конкретных классов будет декорировано одним набором декораторов. Полезным свойством шаблона Декоратор оказывается то, что вы можете добавлять в него столько

конкретных компонентов, сколько пожелаете. Представьте себе большой коммерческий веб-сайт, где конкретным компонентом является его отдельный подсайт для одного определенного вида электронного бизнеса, над которым вы работали несколько месяцев. С помощью шаблона проектирования Декоратор вы разработали несколько полезных и оттестированных элементов, которые были использованы в вашем приложении с помощью разнообразных декораторов. И вскоре после завершения вашего проекта вы получаете заказ на создание еще одного коммерческого подсайта, с другим главным продуктом, но с множеством элементов, часть которых уже разработана вами. В таком случае вместо того, чтобы начинать все с нуля, вам нужно лишь добавить другой класс конкретного компонента и несколько новых декораторов, которых не оказалось в исходном множестве.

Пример 4.3 демонстрирует нам, что класс конкретного компонента делает несколько больше, чем просто расширяет класс `Component` и добавляет конструктор. Он наследует все характеристики класса `Component` и использует переменную `information` для вывода сообщения, указывающего на декоратор, в окно `Output`. Сохраните код из примера 4.3 в файле *ConcreteComponent.as*.

Пример 4.3. ConcreteComponent.as

```
package
{
    // Конкретный компонент
    public class ConcreteComponent extends Component
    {
        public function ConcreteComponent()
        {
            // \u2794 - Unicode-символ стрелки вправо
            information = "Concrete Component is decorated with \u2794";
        }
    }
}
```

В примере 4.3 вы видите, как в строчку вводится символ Unicode с использованием формата `\u` + код символа. В данном примере с помощью кода 2794 формируется символ стрелки. Вы можете найти подобные полезные символы по адресу *http://www.fileformat.info/info/unicode/char/search.htm*. Там вы обнаружите несколько символов стрелок, включая `\u0363`, `\u2192`, `\u21aa` и `\u21d2`.

Следующим нашим шагом будет построение конкретных классов декораций. В данном минималистском примере два конкретных декоратора будут упаковывать в себя конкретный компонент. Поэтому для начала нам потребуется переменная типа `Component` для хранения представителя компонента, который мы и будем упаковывать.

```
var components:Component;
```

Данная переменная называется components, во множественном числе, поскольку слово component является зарезервированным словом в ActionScript 3.0. Эта переменная будет использоваться в методах декоратора. Далее нам нужно как-то присвоить переменной components объект, который будет упаковываться. Следующий код показывает нам, как компонент для упаковки передается конструктору декоратора.

```
public function DecConA(components:Component)
{
    this.components=components;
}
```

И наконец, нам требуется функция-получатель, которая будет предоставлять нам *уникальную* информацию из конкретного декоратора. Публичный метод, унаследованный от класса Decorator, должен быть замещен с использованием выражения override. Наш новый метод получает информацию от конкретного компонента (используя components.getInformation()) и добавляет к ней информацию о конкретном декораторе (+ " Decoration Alpha:"):

```
override public function getInformation():String
{
    return components.getInformation() + " Decoration Alpha:";
}
```

Конкретные декорации

Теперь у нас все готово для написания конкретных классов декораторов. Сохраните примеры 4.4 и 4.5 в файлах *DecConA.as* и *DecConB.as* соответственно.

В некоторых классах декораторов вы увидите выражение:

```
this.components = components;
```

Оно немного нестандартным образом использует идентификатор this, но это имеет смысл в данном контексте, поскольку позволяет различить переменную и параметр метода. Скорее всего, вы не будете использовать идентификатор this таким образом. Но в данном случае имена переменной и параметра идентичны, что было сделано для того, чтобы показать применение абстрактного класса для них обоих.[1]

[1] На самом деле данное использование инструкции this является вполне стандартным, поскольку она ссылается непосредственно на текущий объект, а обращение к переменной внутри функции в первую очередь происходит на локальную переменную и лишь при отсутствии таковой – на свойство объекта. – *Прим. науч. ред.*

Пример 4.4. DecConA.as

```
package
{
    // Конкретный декоратор «Alpha»
    public class DecConA extends Decorator
    {
        private var components:Component;
        public function DecConA(components:Component)
        {
            this.components=components;
        }
        override public function getInformation():String
        {
            return components.getInformation() + " Decoration Alpha:";
        }
    }
}
```

Пример 4.5. DecConB.as

```
package
{
    // Конкретный декоратор «Beta»
    public class DecConB extends Decorator
    {
        var components:Component;
        public function DecConB(components:Component) {
            this.components=components;
        }
        override public function getInformation():String
        {
            return components.getInformation() + " Decoration Beta:";
        }
    }
}
```

Упаковка (Враппер)

Для реализации шаблона проектирования Декоратор главным является знание того, как упаковать компонент в конкретный декоратор. Сначала вам нужно создать представителя конкретного компонента.

```
var testComponent:Component = new ConcreteComponent();
```

Затем упаковать компонент в один или несколько декораторов, используя следующий формат:

```
componentInstance=new ConcreteDecorator(testComponent);
```

Следовательно, в нашем примере с двумя конкретными декораторами мы напишем:

```
testComponent=new DecConA(testComponent);
testComponent=new DecConB(testComponent);
```

Таким образом, `testComponent` будет декорирован двумя декораторами. Мы можем повторить вышеуказанные строчки столько раз, сколько захотим. Представьте себе, что декораторы – это красные и зеленые украшения на новогоднюю елку. Вы повесите на нее скорее украшения обоих цветов, чем только одного. Шаблон проектирования Декоратор работает в этом смысле кумулятивно. То есть очередной дополнительный декоратор добавляется к тем, что уже упаковывают в себя конкретный компонент.

И наконец, чтобы посмотреть, какой хаос мы произвели, мы используем метод-получатель `getinformation()`:

```
trace(testComponent.getInformation( ));
```

Чтобы увидеть, как это все работает, сохраните пример 4.6 в файле ActionScript *DecTest.as*. Затем откройте новый документ **Flash и введите** `DecTest` в его поле класса на панели свойств.

Пример 4.6. DecTest.as

```
package
{
    import flash.display.Sprite;
    public class DecTest extends Sprite
    {
        public function DecTest()
        {
            // Реализуем конкретный компонент
            var testComponent:Component = new ConcreteComponent();
            // Упаковываем ее в первый декоратор
            testComponent=new DecConA(testComponent);
            // Упаковываем ее во второй декоратор
            testComponent=new DecConB(testComponent);
            // Выводим результаты
            trace(testComponent.getInformation());
        }
    }
}
```

Рисунок 4.5 показывает, что вы должны увидеть в окне `Output`, когда запустите это приложение.

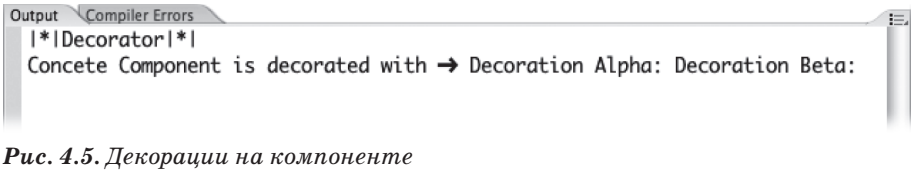


Рис. 4.5. *Декорации на компоненте*

Чтобы понять, что же произошло в нашей реализации шаблона Декоратор, вернемся немного назад и посмотрим на рис. 4.2. Наше приложение сначала создает объект `ConcreteComponent()`. Он показывает сообщение, указывающее на его декорации. Представьте себе этот объект (`testComponent`) самой маленькой матрешкой слева на рис. 4.2. Данная матрешка помещается в матрешку-декоратор №1. Теперь объект конкретного компонента (`testComponent`) декорирован матрешкой №1, но у него остались его исходные свойства. Подобным образом домашний газон, украшенный семейными гномами, все равно остается покрытым зеленой травой. Далее матрешка №1, которая уже содержит конкретный компонент, помещается в матрешку №2. Теперь матрешка №2 содержит в себе матрешку №1 и конкретный компонент. Следовательно, матрешка №2 имеет все свои собственные свойства плюс свойства всего того, что у нее внутри.

В известном смысле весь этот процесс работает, как составной оператор «плюс-равно» (`+=`). Каждый новый декоратор добавляет себя к уже существующим компоненту и его декораторам. По мере добавления каждого декоратора все предыдущие декораторы остаются, но не дублируются, если только вы не добавите один декоратор более одного раза. Итак, как показывает нам итоговое окно, вы можете добавлять столько декораторов, сколько хотите, просто упаковывая их как единый объект в следующий декоратор.

Применение простого шаблона Декоратор во Flash: картонная кукла

Мы можем перейти от концептуального примера к более конкретному с помощью компьютеризированной игры с картонной куклой. Компонентом этой игры будет кукла, которую нужно одеть, а декораторами – части одежды для куклы. Используя новый класс ActionScript 3.0 `Loader`, мы будем декорировать различными *gif*-файлами исходное *gif*-изображение викторианской картонной куклы. Как и в предыдущем абстрактном примере, здесь у нас будет один конкретный компонент и несколько декораций.

В данном конкретном примере мы не будем создавать интерфейс пользователя, чтобы не отвлекаться от использования шаблона Декоратор в приложении. Но мы получим итоговое графическое изображение, на котором увидим результат работы нашего шаблона. (Далее в этой главе у нас будет пример с полным интерфейсом пользователя.)

Создание класса компонента

Первым нашим классом является абстрактный класс компонента `Model`. Все последующие классы будут его подклассами. Он очень прост, но крайне важен для всего нашего приложения. Как вы увидите, он весьма схож с предыдущим нашим абстрактным компонентом. Он состоит из

метода `getDressed()` («одеть») и строковой переменной `whatToWear` («во что одеть»), как показано в примере 4.7. Сохраните его в файле *Model.as*.

Пример 4.7. Model.as

```
package
{
    // Абстрактный класс
    public class Model
    {
       protected var whatToWear:String;
        public function getDressed():String
        {
            return whatToWear;
        }
    }
}
```

Не забудьте, что класс из примера 4.7 должен рассматриваться как абстрактный. Так что метод `getDressed()` должен быть замещаемой публичной функцией, чтобы работать так, как надо. Однако свойство `whatToWear` не нуждается в модификации своих наследуемых характеристик.

Класс декоратора: одежда для куклы

Пример 4.8 содержит в себе абстрактный класс декоратора `Dresser` (дословно «одевальщик»), который расширяет класс компонента `Model`. Основным вкладом здесь является новая имплементация метода `getDressed()` с использованием унаследованного свойства `whatToWear`.

Пример 4.8. Dresser.as

```
package
{
    // Абстрактный класс
    public class Dresser extends Model
    {
        override public function getDressed():String
        {
            return whatToWear;
        }
    }
}
```

Все конкретные декораторы будут подклассами этого класса.

Конкретные классы

Итак, два основных абстрактных класса шаблона Декоратор готовы; теперь вам необходимо создать конкретные классы. В предыдущем минималистском примере Декоратора вы видели результаты работы благодаря применению функции `trace()`. В конкретном компоненте и в деко-

раторах вместо функций `trace()` нам придется использовать некоторое форматирование для дальнейшего разбора строки; теперь в них будет добавляться знак тильда (~) как символ разделителя. Он будет нам необходим, поскольку все строки из конкретного компонента и из декораторов будут собираться в одну большую строку для дальнейшего использования.

Класс конкретного компонента

Класс конкретного компонента является единственным классом, происходящим напрямую от исходного абстрактного класса. Все остальные классы в этом приложении будут расширять абстрактный класс декоратора. Используя простую функцию конструктора `Sue()`, данный класс присваивает значение переменной `whatToWear`. Этого достаточно, чтобы идентифицировать данный класс как представителя главного абстрактного класса компонента `Model` и дать ему уникальное имя. Все декораторы используют конкретный компонент как основу для своих декораций. Сохраните пример 4.9 в файле *Sue.as*.

Пример 4.9. Sue.as

```
package
{
    public class Sue extends Model
    {
        public function Sue()
        {
            whatToWear="~sue";
        }
    }
}
```

Если вы хотите иметь больше классов конкретных компонентов, вам нужно лишь создать подобные классы с другими именами и другими значениями для переменной `whatToWear`. При такой их структуре не существует ограничений на число новых конкретных компонентов, которые вы можете добавить в приложение.

Классы конкретных декораторов

Отталкиваясь от нашего минималистского примера использования Декоратора, показанного ранее в этой главе, можно отметить, что понять работу данного шаблона легче, если результаты его работы видны на графическом экране. Когда создается начальный представитель конкретного компонента, все ссылки в конкретных классах декораторов устанавливаются на него. (См. пример 4.16, где конкретный компонент упаковывается в декораторы.) Во всех наших конкретных классах декораторов ссылка из переменной `model` должна быть ссылкой на объект конкретного компонента. В данном случае это представитель класса `Sue`, но он может быть представителем и любым другим конкретным

компонентом. Вот почему, если вы захотите расширить ваше приложение и включить в него больше конкретных компонентов (других картонных кукол для одевания), вам не придется проводить никаких фундаментальных изменений; вам потребуется только добавить еще один класс конкретного компонента.

В примерах с 4.10 по 4.15 в заголовках указаны имена файлов, в которых их следует сохранить.

Пример 4.10. OrangeDress.as

```
package
{
    public class OrangeDress extends Dresser
    {
        private var model:Model;
        public function OrangeDress(model:Model)
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~orangedress";
        }
    }
}
```

Пример 4.11. BlueDress.as

```
package
{
    public class BlueDress extends Dresser
    {
        private var model:Model;
        public function BlueDress(model:Model)
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~bluedress";
        }
    }
}
```

Пример 4.12. Bow.as

```
package
{
    public class Bow extends Dresser
    {
        private var model:Model;
        public function Bow(model:Model)
```

```
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~bow";
        }
    }
}
```

Пример 4.13. Umbrella.as

```
package
{
    public class Umbrella extends Dresser
    {
        private var model:Model;
        public function Umbrella (model:Model)
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~ umbrella ";
        }
    }
}
```

Пример 4.14. Hat.as

```
package
{
    public class Hat extends Dresser
    {
        private var model:Model;
        public function Hat(model:Model)
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~hat";
        }
    }
}
```

Пример 4.15. Muff.as

```
package
{
    public class Muff extends Dresser
    {
        private var model:Model;
```

```
        public function Muff(model:Model)
        {
            this.model=model;
        }
        override public function getDressed():String
        {
            return model.getDressed() + "~muff";
        }
    }
}
```

Все шесть декораторов могут быть использованы в любой желаемой комбинации с представителем класса `Model`. В следующем разделе показано, как имплементировать шаблон проектирования Декоратор со многими различными декораторами.

Имплементация декоратора для картонной куклы

Как уже отмечалось, для данного приложения не будет создаваться какой-либо интерфейс пользователя, поэтому вы сможете ясно увидеть, как конкретный компонент упаковывается выбранными декораторами. К тому же на сцене вы сможете увидеть эффекты от их различных комбинаций.

Наша последовательность действий состоит в следующем:

1. Создаем представителя конкретного компонента.
2. Упаковываем конкретный компонент в представителя выбранного декоратора, используя следующий формат:

```
componentInstance = new DecoratorInstance(componentInstance)
```

3. Применяем метод-получатель (`getDressed()`) к представителю конкретного компонента, что даст нам полностью упакованные значения. Они будут включать в себя значение из конкретного компонента и значения из всех примененных декораторов.

Пример 4.16 показывает нам, как реализуется Декоратор для картонной куклы.

Пример 4.16. FashionShow.as

```
package
{
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.display.Sprite;
    public class FashionShow extends Sprite
    {
        var ensemble:Array=new Array();
        public function FashionShow()
        {
            trace("||--Working--||");
```

```
            var doll:Model = new Sue();
            doll=new Hat(doll);
            doll=new OrangeDress(doll);
            //doll=new BlueDress(doll);
            //doll=new Umbrella(doll);
            doll=new Bow(doll);
            doll=new Muff(doll);
            var ready2wear:String=doll.getDressed();
            sewingMachine(ready2wear);
            for (x=0; x < ensemble.length; x++)
            {
                clothesOn(ensemble[x]);
            }
        }
        private function sewingMachine(wardrobe:String):Array
        {
            ensemble=wardrobe.split("~");
            ensemble.shift();
            return ensemble;
        }
        private function clothesOn(outfit:String)
        {
            var clothier:Loader=new Loader();
            var item:String="clothes/" + outfit + ".gif";
            var getItem:URLRequest=new URLRequest(item);
            clothier.load(getItem);
            this.addChild(clothier);
        }
    }
}
```

Когда все вышеназванное у вас будет готово, вам потребуются еще несколько *gif*-файлов. Нашей моделью (конкретным компонентом) будет изображение куклы, которую вы будете одевать с помощью шаблона Декоратор. Каждое имя *gif*-файла должно совпадать со значением, добавляемым переменной `whatToWear` в коде конкретного компонента или декоратора, за исключением символа тильда (~). Например, в конкретном компоненте `Sue` переменной `whatToWear` присваивается строковое значение "`~sue`", и, значит, *gif*-файлом для этого компонента будет *sue.gif*. Декоратор `BlueDress` («голубое платье») добавляет `whatToWear` значение "`~bluedress`", следовательно, файл с изображением голубого платья должен будет называться *bluedress.gif*. Поместите все *gif*-файлы, включая файл с изображением куклы, в каталоге под названием `clothes`.

И наконец, откройте новый документ **Flash и установите для него размер сцены 300×650**. В панели свойств в поле класса этого документа введите `FashionShow`. Размеры и пропорции вашей сцены будут зависеть от размеров изображения картонной куклы, которое вы будете использовать.

Рисунок 4.6 показывает два различных варианта комбинаций декораторов на картонной кукле. Слева ее изображение декорировано шляпой, бантом, оранжевым платьем и муфтой, а справа – голубым платьем, бантом и зонтиком. Они так сильно непохожи друг на друга всего лишь из-за различных наборов декораторов.

Рис. 4.6. *Картонная кукла с двумя различными наборами декораций*

Вместо картонных кукол вы можете декорировать любой компонент любыми новыми свойствами. Кроме того, вы можете включать более одного свойства или метода в декорацию. В следующем разделе мы рассмотрим добавление дополнительных свойств с помощью шаблона Декоратор.

Декорация смертными грехами и райскими добродетелями

В боевых компьютерных играх друг против друга воюют различные виды героев и злодеев, и каждый из противников имеет различное вооружение и защиту. То есть они *декорированы* различными характеристиками и возможностями. Что может быть лучше для примера нового функционала в шаблоне Декоратор, чем столкновение сил добра и зла?

В табл. 4.1 перечислены смертные грехи и райские добродетели. (Этот список мы почерпнули из *«Inferno»* Данте и эпической поэмы *«Psychomachia»* Пруденция, хотя оба они, как мы полагаем, использовали примитивные персональные компьютеры Commodore-64 для составления своих списков.)

Таблица 4.1. Декорации добром и злом

Смертный грех	Описание	Добродетель	Описание
Гнев	Неконтролируемый гнев – массовые синтаксические ошибки в программе.	Сострадание	Забота об остальных – помощь процедурным программистам в переходе на ООП.
Злоба	Мстительность, злорадство, недоброжелательность, жестокость, ненависть к другим – непочтительные замечания о Linux.	Храбрость	Правильные поступки, несмотря на все опасности – использование объектно-ориентированного программирования.
Запутывание	Сокрытие правды – искажение фактов добавлением к ним беспорядка – программирование без комментариев.	Надежда	Вера в неизбежность победы добра над злом – вы действительно можете закончить проект вовремя.
Высокомерие	Непомерная гордость, презрение к чувствам, знаниям или вере других – вера в то, что MS Windows является единственной настоящей ОС.	Справедливость	Строгий баланс и равные шансы для всех – использование Windows, Mac OS и Linux.
Предвзятость	Осуждение других на основе стереотипов, а не их действий – поддразнивание пользователей Макинтошей.	Открытость	Способность принимать новые знания, идеи и противоположные идеи – написание программы на Макинтоше.

Смертный грех	Описание	Добродетель	Описание
Догматизм	Жесткая, упертая вера, даже несмотря на признаки обратного – продолжение использования процедурного программирования.	Целостность	Верность своим ценностям даже при соблазне оставить их ради сиюминутной выгоды – отказ от грязных трюков в коде несмотря на то, что они решают проблему и заказчик о них никогда не узнает.
Безразличие	Бездействие при виде страданий, неоказание помощи – нежелание помочь другим в изучении ООП.	Усердие	Желание взяться за особенно сложную задачу, чтобы ее выполнить, – изучение шаблонов проектирования.

В качестве примеров для описания смертных грехов и райских добродетелей первыми приходят на ум некоторые качества и действия, встречавшиеся нам в данной главе. Поставленная перед нами задача достаточно проста, поскольку, как и в примере с картонной куклой, нам нужно всего лишь приписать определенные качества каждому из декораторов. Однако это еще далеко не все, что мы можем сделать с помощью шаблона проектирования Декоратор, в чем вы сможете убедиться в следующих двух разделах.

Добавление свойств и методов

До сих пор мы использовали в базовом абстрактном классе компонента одну переменную и один метод-получатель. Однако, как и любой другой класс, эта базовая структура может содержать гораздо больше, чем одну переменную и функцию. Сохраните нижеследующий пример в файле *Component.as*.

Пример 4.17. Component.as

```
package
{
    // Абстрактный класс
    public class Component
    {
        protected var soul:String="All that is inside a spirit";
        protected var goodness:Number;
        protected var vice:Number;

        public function getSoul():String
        {
            return soul;
```

```
        }
        public function good():Number
        {
            return goodness;
        }
        public function evil():Number
        {
            return vice;
        }
    }
}
```

Как и в предыдущем примере, мы начинаем с создания свойства `soul` («душа»). (Ему присваивается начальное строковое значение `"All that is inside a spirit"` («Все, что внутри – это дух»), но это в действительности не важно, поскольку оно представляет собой абстрактный класс, и никто его значения не увидит и не будет использовать, а сделано только для ясности.) Далее определяются два числовых свойства – `goodness` («добродетель») и `vice` («порок»). Они будут аккумулировать в себе все добро и зло данной души.

Три функции-получателя предназначены для выдачи значений этой строки и двух числовых переменных. Теперь наш абстрактный класс компонента полностью готов к работе.

Несколько конкретных компонентов

Мы можем, конечно, следить только за одной душой, но что в этом интересного? Более того, в большинстве приложений использование только одного компонента для декорирования приносит мало пользы. Поэтому мы займемся декорированием не одной души, а сразу двух. До поры до времени мы забудем обо всех дебатах, связанных с первородным грехом, и будем создавать наши души в исходном чистом состоянии. И `goodness`, и `vice` будут изначально устанавливаться в ноль. Просто дадим разные значения переменной `soul`, чтобы различать наши души по именам.

Примеры 4.18 и 4.19 содержат в себе конкретные компоненты для классов `Dick` («Дик») и `Jane` («Джейн»). В каждом классе нам нужно присвоить строковое значение для переменной `soul` и два числовых значения для `goodness` и `vice` – всего того, что было унаследовано нами из класса `Component`. Сохраните пример 4.18 как *Dick.as*, а пример 4.19 как *Jane.as*.

Пример 4.18. Dick.as

```
package
{
    public class Dick extends Component
    {
```

```
        public function Dick()
        {
            soul = "Dick's soul\n";
            goodness=0;
            vice=0;
        }
    }
}
```

Пример 4.19. Jane.as

```
package
{
    public class Jane extends Component
    {
        public function Jane()
        {
            soul = "Jane's soul\n";
            goodness=0;
            vice=0;
        }
    }
}
```

Данные классы конкретных компонентов очень похожи на классы конкретных компонентов из предыдущих примеров, но теперь у нас их два, а не один. К тому же, вместо того чтобы использовать только одну строковую переменную, мы добавили к ней две числовые переменные со своими значениями. В остальном данные классы точно такие же.

В обоих данных конкретных классах свойства наследуются непосредственно от абстрактного класса `Component`. Никаких замещений или специальных настроек не требуется, потому что `Dick` и `Jane` — это обычные компоненты, которые будут декорироваться с помощью классов-декораторов. Все замещения и дополнения, дающие конкретным декораторам их уникальные свойства и методы, создаются *классами декораторов*. Поэтому изменения в классах конкретных компонентов не нужны.

Декорирование несколькими свойствами

Насколько просто добавить несколько свойств и методов компонентам, настолько же просто добавить их классам декораторов. Вместо того чтобы использовать только одно свойство или метод, вы повторяете все то же самое для нескольких методов или классов. Пример 4.20 содержит абстрактный класс `Decorator`, являющийся подклассом класса `Component`. Сохраните его в файле *Decorator.as*.

Пример 4.20. Decorator.as

```
package
{
```

```
    // Абстрактный класс
    public class Decorator extends Component
    {
        override public function getSoul():String
        {
            return soul;
        }
        override public function good():Number
        {
            return goodness;
        }
        override public function evil():Number
        {
            return vice;
        }
    }
}
```

Как подкласс класса `Component` абстрактный класс `Decorator` не делает ничего другого, как просто реимплементирует функции-получатели: одну – для получения строки, а две другие – для получения чисел. Возвращаемые таким образом свойства были изначально определены в классе `Component`, и, поскольку класс `Decorator` является его подклассом, ему не нужно имплементировать их заново. Однако, как и в предыдущих примерах, их функции-получатели ре-имплементируются. Разница лишь в том, что теперь их больше. Логика же всего процесса осталась прежней.

Множественные методы конкретных декораторов

Когда дело доходит до конкретных декораторов данного учебного приложения, мы видим нечто новое. Для начала обсудим пример 4.21. Он является только общей иллюстрацией, и его *не* нужно сохранять в виде кода. Просто посмотрите на него. В рабочем коде его значения будут заменены некоторыми конкретными элементами.

Пример 4.21. Обобщенные конкретные декорации

```
package
{
    // Общий пример – не имплементируется
    public class GoodEvil extends Decorator
    {
        private var components:Component;
        public function GoodEvil(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|GoodEvil";
```

```
        }
        override public function good():Number
        {
            return +/-00 + components.good();
        }
        override public function evil():Number
        {
            return +/-00 + components.evil();
        }
    }
}
```

Если рассматривать пример 4.21 по частям, то его первая часть точно такая же, как и в предыдущих наших примерах приложений. Создается представитель класса `Component` с именем `components`, и конструктор присваивает объект `components` сам себе:

```
var components:Component;
public function GoodEvil(components:Component)
{
    this.components=components;
}
```

Далее ре-имплементируется функция-получатель `getSoul()` с целью получения от нее текущего значения всех декораций конкретного компонента с добавлением к нему собственного значения для декорации. И снова это подобно тому, что делалось в предыдущих примерах.

```
override public function getSoul():String
{
    return components.getSoul() + "|GoodEvil";
}
```

Следующие две функции добавляют или вычитают числовые значения с помощью методов `good()` («добро») и `evil()` («зло»). Каждый хороший поступок служит добру и уменьшает зло, а пороки служат злу и вычитаются из добра. Поэтому в зависимости от конкретного декоратора вы будете добавлять или вычитать что-то из каждого из двух возвращаемых значений и добавлять это к текущему значению конкретного компонента.

```
override public function good():Number
{
    return +/-00 + components.good();
}
override public function evil():Number
{
    return +/-00 + components.evil();
}
```

Примеры с 4.22 по 4.35 представляют собой действие семи смертных (отредактированных) грехов и семи райских (отредактированных) до-

бродетелей. Но на самом деле они просто иллюстрируют нам всю гибкость шаблона проектирования Декоратор. (Равно как и всю сложность темы спасения души.)

Конкретные декораторы добра и зла

Ниже представлены 14 классов конкретных декораторов. Все они очень похожи и отличаются друг от друга лишь своими именами и значениями, присваиваемыми числовым свойствам. Вам будет гораздо проще, создав один из них, копировать его в новый файл ActionScript и редактировать, чем набивать каждый файл с чистой страницы. Закончив создание этих 14 классов, не останавливайтесь и добавьте еще парочку ваших собственных декораторов добра и зла. В примерах с 4.22 по 4.35 в заголовке указано имя файла для сохранения.

Райские добродетели

Пример 4.22. Integrity.as

```
package
{
    public class Integrity extends Decorator
    {
        private var components:Component;
        public function Integrity(components:Component) {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Integrity";
        }
        override public function good():Number
        {
            return 14 + components.good();
        }
        override public function evil():Number
        {
            return -6 + components.evil();
        }
    }
}
```

Пример 4.23. Hope.as

```
package
{
    public class Hope extends Decorator
    {
        private var components:Component;
        public function Hope(components:Component)
        {
            this.components=components;
```

```
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Hope";
        }
        override public function good():Number
        {
            return 5 + components.good();
        }
        override public function evil():Number
        {
            return -10 + components.evil();
        }
    }
}
```

Пример 4.24. Courage.as

```
package
{
    public class Courage extends Decorator
    {
        private var components:Component;
        public function Courage(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Courage";
        }
        override public function good():Number
        {
            return 10 + components.good();
        }
        override public function evil():Number
        {
            return -8 + components.evil();
        }
    }
}
```

Пример 4.25. Compassion.as

```
package
{
    public class Compassion extends Decorator
    {
        private var components:Component;
        public function Compassion(components:Component)
        {
            this.components=components;
        }
```

```
        override public function getSoul():String
        {
            return components.getSoul() + "|Compassion";
        }
        override public function good():Number
        {
            return 7 + components.good();
        }
        override public function evil():Number
        {
            return -15 + components.evil();
        }
    }
}
```

Пример 4.26. Openness.as

```
package
{
    public class Openness extends Decorator
    {
        private var components:Component;
        public function Openness(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Openness";
        }
        override public function good():Number
        {
            return 12 + components.good();
        }
        override public function evil():Number
        {
            return -15 + components.evil();
        }
    }
}
```

Пример 4.27. Diligence.as

```
package
{
    public class Diligence extends Decorator
    {
        private var components:Component;
        public function Diligence(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
```

```
        {
            return components.getSoul() + "|Diligence";
        }
        override public function good():Number
        {
            return 10 + components.good();
        }
        override public function evil():Number
        {
            return -5 + components.evil();
        }
    }
}
```

Пример 4.28. Justice.as

```
package
{
    public class Justice extends Decorator
    {
        private var components:Component;
        public function Justice(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Justice";
        }
        override public function good():Number
        {
            return 9 + components.good();
        }
        override public function evil():Number
        {
            return -9 + components.evil();
        }
    }
}
```

Смертные грехи

Пример 4.29. Rage.as

```
package
{
    public class Rage extends Decorator
    {
        private var components:Component;
        public function Rage(components:Component)
        {
            this.components=components;
        }
```

```
        override public function getSoul():String
        {
            return components.getSoul() + "|Rage";
        }
        override public function good():Number
        {
            return -9 + components.good();
        }
        override public function evil():Number
        {
            return 8 + components.evil();
        }
    }
}
```

Пример 4.30. Malice.as

```
package
{
    public class Malice extends Decorator
    {
        private var components:Component;
        public function Malice(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Malice";
        }
        override public function good():Number
        {
            return -14 + components.good();
        }
        override public function evil():Number
        {
            return 12 + components.evil();
        }
    }
}
```

Пример 4.31. Prejudice.as

```
package
{
    public class Prejudice extends Decorator
    {
        private var components:Component;
        public function Prejudice(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
```

```
        {
            return components.getSoul() + "|Prejudice";
        }
        override public function good():Number
        {
            return -10 + components.good();
        }
        override public function evil():Number
        {
            return 15 + components.evil();
        }
    }
}
```

Пример 4.32. Obfuscation.as

```
package
{
    public class Obsfuscation extends Decorator
    {
        private var components:Component;
        public function Obsfuscation(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Obsfuscation";
        }
        override public function good():Number
        {
            return -12 + components.good();
        }
        override public function evil():Number
        {
            return 7 + components.evil();
        }
    }
}
```

Пример 4.33. Dogmatisms.as

```
package
{
    public class Dogmatisms extends Decorator
    {
        private var components:Component;
        public function Dogmatisms(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
```

```
            return components.getSoul() + "|Dogmatisms";
        }
        override public function good():Number
        {
            return -12 + components.good();
        }
        override public function evil():Number
        {
            return 15 + components.evil();
        }
    }
}
```

Пример 4.34. Arrogance.as

```
package
{
    public class Arrogance extends Decorator
    {
        private var components:Component;
        public function Arrogance(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
            return components.getSoul() + "|Arrogance";
        }
        override public function good():Number
        {
            return -5 + components.good();
        }
        override public function evil():Number
        {
            return 5 + components.evil();
        }
    }
}
```

Пример 4.35. Indifference.as

```
package
{
    public class Indifference extends Decorator
    {
        private var components:Component;
        public function Indifference(components:Component)
        {
            this.components=components;
        }
        override public function getSoul():String
        {
```

```
            return components.getSoul() + "|Indifference";
        }
        override public function good():Number
        {
            return -9 + components.good();
        }
        override public function evil():Number
        {
            return 10 + components.evil();
        }
    }
}
```

На этом этапе вы можете добавить ваши собственные декораторы. Используйте для них тот же формат, что и для остальных. Вы можете добавить также дополнительные конкретные компоненты. То есть помимо Дика и Джейн у вас могут быть другие персонажи, которых вы бы хотели отдекорировать добром и злом.

Имплементация декораторов добра и зла

Вместо одной имплементации мы попробуем создать две. Первая из них будет видоизменять два разных конкретных компонента и выводить результаты своей работы в окно Output. Вторая будет забирать эти результаты, использовать их для размещения микроклипов на «графике душ» и добавлять строковые метки микроклипам ангела и демона в зависимости от того, что из добра и зла преобладает.

Сдвоенная имплементация

Наша первая имплементация декорирует два конкретных компонента Dick и Jane. Она будет использовать все 14 смертных грехов и райских благодетелей, но вы можете использовать любую их комбинацию, которую только захотите. По мере декорирования каждого конкретного компонента (life («жизнь») и light («свет»)) свойства good и evil увеличиваются или уменьшаются, в зависимости от того, какой декоратор применяется к конкретному компоненту. Ссылка на объект components в каждом из декораторов является ссылкой на конкретный компонент для декорирования. И с каждым новым декоратором ее значение поднимается или опускается, в зависимости от этого декоратора. Сохраните пример 4.36 в файле ActionScript *MainDual.as*. Введите MainDual в поле класс документа на панели свойств файла *Dual.fla* и сохраните его. Теперь протестируйте данное приложение.

Пример 4.36. MainDual.as

```
package
{
    import flash.display.Sprite;
    public class MainDual extends Sprite
```

```
    {
        public function MainDual()
        {
            trace("||--Judging--||");
            var life:Component=new Dick();
            var light:Component=new Jane();
            // Добавляем добро
            life=new Courage(life);
            light=new Hope(light);
            light=new Compassion(light);
            life=new Openness(life);
            life=new Diligence(life);
            light=new Justice(light);
            // Добавляем зло
            light=new Malice(light);
            life=new Prejudice(life);
            life=new Dogmatisms(life);
            life=new Arrogance(life);
            life=new Indifference(life);
            life=new Rage(life);
            light=new Obsfuscation(light);
            trace(life.getSoul()+"\nTotal Virture: "+
                (life.good()-life.evil()));
            trace(light.getSoul()+"\nTotal Virture: "+
                (light.good()-light.evil()));
        }
    }
}
```

На выходе всего этого добра и зла появится:

```
||--Judging--||
Dick's soul
|Courage|Openness|Diligence|Prejudice|Dogmatisms|Arrogance
    |Indifference|Rage
Total Virture: -38
Jane's soul
|Hope|Compassion|Justice|Malice|Obsfuscation
Total Virture: 10
```

Поскольку в выражении `trace()` из суммарного значения метода `good()` вычитается результат метода `evil()`, любой положительный результат говорит о преобладании хороших качеств, а негативный – об избытке негативных черт характера.

Вы можете несколько раз упаковывать один конкретный компонент одним и тем же декоратором. (Например, тот, кто думает, что сможет вовремя закончить отстающий на месяц по срокам выполнения проект, не поспав одну недельку, вдвойне самонадеян, втройне усерден и, возможно, к тому же вдвойне высокомерен.)

Ранжирование душ

Как и в случае применения любого другого шаблона проектирования, то, что вы делаете с результатом его работы, зависит от вас. Но, поскольку ActionScript 3.0 является частью Flash, в следующей имплементации мы покажем вам, как выводить результат в различных форматах с помощью графических элементов. Метод `getSoul()` генерирует для нас строку, а `good()` и `evil()` – числа. Строка будет помещаться в текстовое поле, встроенное в клип, а вертикальная позиция этого клипа будет определяться значениями, генерируемыми методами `good()` и `evil()`. Чтобы начать работу, сохраните код из примера 4.37 в файле ActionScript под именем *Soul.as*.

Пример 4.37. Soul.as

```
package
{
    import flash.display.Sprite;
    import flash.display.MovieClip;
    public class Soul extends Sprite
    {
        // Реализуем два мувиклипа
        // из Библиотеки
        var devil:Devil=new Devil();
        var angel:Angel=new Angel();
        public function Soul()
        {
            var life:Component=new Jane();
            // Добавляем добро
            life=new Courage(life);
            life=new Compassion(life);
            life=new Hope(life);
            //life=new Integrity(life);
            life=new Openness(life);
            life=new Diligence(life);
            life=new Justice(life);
            // Добавляем зло
            life=new Malice(life);
            life=new Prejudice(life);
            //life=new Dogmatisms(life);
            life=new Arrogance(life);
            //life=new Indifference(life);
            //life=new Rage(life);
            //life=new Obsfuscation(life);
            setAngelDevil(life.good(),life.evil(),life.getSoul());
        }
        private function setAngelDevil(right:Number,wrong:Number,
            eternalsoul:String)
        {
```

```
                this.addChild(devil);
                this.addChild(angel);
                var booWrong:Number=Number(wrong>0);
                var booRight:Number=Number(right>0);
                devil.x=330;
                devil.y=270-((wrong*booWrong)*(270/72));
                angel.x=96;
                angel.y=270-((right*booRight)*(270/60));
                if (booWrong)
                {
                    devil.soul_txt.text=eternalsoul;
                } else
                {
                    angel.soul_txt.text=eternalsoul;
                }
            }
        }
    }
```

Данный скрипт просто передает три значения, сгенерированные методами-получателями `good()`, `evil()` и `getSoul()`, функции `setAngelDevil()`. Функция `setAngelDevil()` использует два клипа из библиотеки и позиционирует их на сцене. В зависимости от показателей имя конкретного компонента появляется на фоне изображения ангела или демона. На рис. 4.7 показано, как это выглядит. Используйте его в качестве образца для описываемого ниже процесса конфигурации вашей сцены.

Следующие шаги помогут вам подготовить два класса `MovieClip` в среде разработки Flash и сконфигурировать сцену для «ранжирования душ».

1. Откройте новый документ Flash, введите `Soul` в его поле класса и сохраните его как *SoulChart.fla*.
2. Добавьте еще один уровень к существующему и назовите верхний уровень `Labels`, а нижний – `Lines`.
3. Выберете из меню Insert → New Symbol. Введите в поле Name слово `Angel` и поставьте метку напротив Export for ActionScript. Кликните на OK. Так вы перейдете в режим редактирования. (Убедитесь, что первая буква в слове «Angel» заглавная, ведь это имя класса.)
4. В окне редактирования символа нарисуйте или скопируйте в него изображение ангела.
5. Кликните на иконку Text на панели инструментов и выберите в качестве типа текста Dynamic Text. На панели свойств для этого объекта динамического текста наберите имя `soul_txt`.
6. Выйдите из режима редактирования символа, кликнув на иконке Scene 1. Теперь вы должны увидеть на панели Library иконку клипа с именем «Angel».

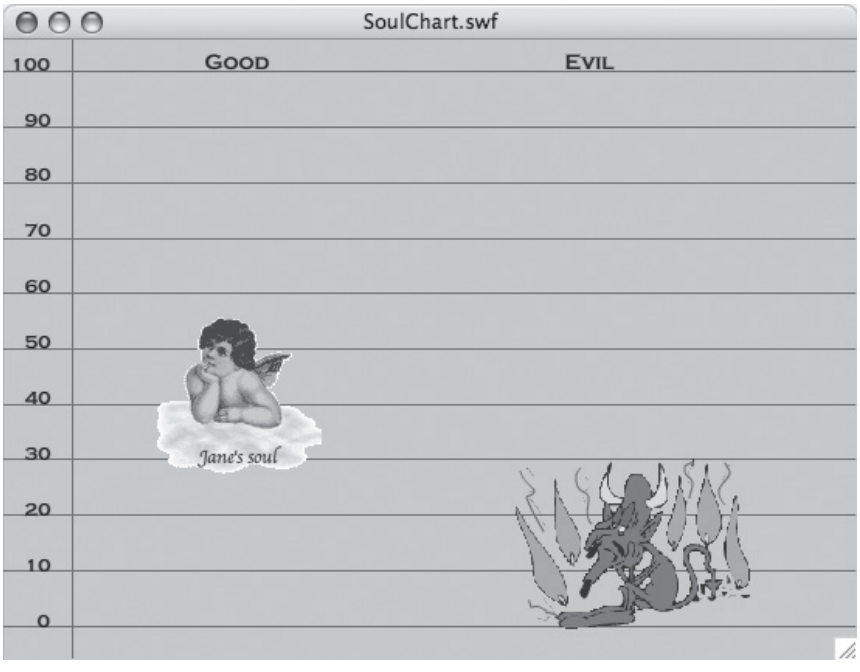


Рис. 4.7. *Использование генерируемых декораторами значений для размещения и маркировки*

7. Повторите шаги с 3-го по 6-ой, заменив слова «Angel» на «Devil». Когда вы закончите, на панели Library вы должны увидеть иконки клипов Ангела и Демона.
8. Кликните на уровне с именем Lines и нарисуйте на нем 11 горизонтальных линий и одну вертикальную, как изображено на рис. 4.7. Заблокируйте этот уровень.
9. Кликните на уровне Labels и пронумеруйте на нем линии от 0 до 100, а также поместите метки «Good» (Добро) и «Evil» (Зло) вверху сцены, как показано на рисунке 4.7. Заблокируйте этот уровень и сохраните ваш файл еще раз.

Добавляя и удаляя символы комментирования перед строками в файле *Soul.as*, вы можете изменять вертикальные позиции изображений ангела и демона. Возможно, вам придется сделать некоторые поправки в зависимости от размеров ваших клипов Ангела и Демона и/или изменить размер по умолчанию вашей сцены со значения в 550 пикселов на 400.

Динамический выбор компонентов и декораций: салон гибридных автомобилей

До настоящего момента наши примеры были обращены на различные аспекты шаблона проектирования Декоратор с акцентом на то, как различные элементы этого шаблона могут быть использованы с разными компонентами и декорациями. Мы применяли различные варианты представления результатов работы приложений, но у нас не было примера с динамическим вводом данных пользователем. Данный пример шаблона Декоратор будет обеспечивать выбор автомобилей и их опций для иллюстрации того, как использовать динамически вводимые данные для декораторов и конкретных компонентов.

Представьте, что вам нужно создать и поддерживать веб-сайт для автосалона. В этом бизнесе каждый год появляются новые модели, и становятся доступными новые опции. Вы никогда не будете знать заранее, какие опции появятся или исчезнут и будут ли текущие модели машин продаваться в следующем году. Вы решаете использовать для вашего приложения шаблон Декоратор, поскольку с его помощью вы можете легко добавить или удалить конкретные компоненты и конкретные декораторы. Конкретными компонентами в данном случае будут различные модели автомобилей, а опции к ним будут различными декораторами. Какая бы новая модель не появилась, вы просто создадите его конкретный компонент, чтобы он ей полностью удовлетворял. Такая же ситуация и с опциями для каждой из моделей; все, что вам нужно будет изменить, – это конкретные декораторы. Вы можете легко добавлять новые или изменять существующие декорации без замены базовой структуры программы.

Создание компонентов для гибридных машин

Данный автосалон решил специализироваться на продаже четырех видов гибридных машин. Для нашего примера мы будем использовать следующие их виды: Тойота Приус, Меркури Маринер, Форд Эскейп и Хонда Аккорд. Следовательно, данный пример потребует от нас создания, помимо исходного класса абстрактного компонента, еще четырех классов конкретных компонентов.

Абстрактный компонент автомобиля

Для начала создадим класс абстрактного компонента автомобиля с именем `Auto`. Ему нужны только строчка для названия автомобиля и числовая переменная для его базовой цены. Две функции-получатели типа автомобиля и его цены завершают создание этого класса. Сохраните код из примера 4.38 в файле *Auto.as*.

Пример 4.38. Auto.as

```
package
{
    // Абстрактный класс
    public class Auto
    {
        protected var information:String;
        protected var bucks:Number;

        public function getInformation():String
        {
            return information;
        }
        public function price():Number
        {
            return bucks;
        }
    }
}
```

Как и всегда в шаблоне проектирования Декоратор, все остальные классы будут наследниками этого класса абстрактного компонента. В следующем разделе вы увидите, что все классы конкретных компонентов тоже являются подклассами класса `Auto`.

Классы конкретных компонентов гибридных машин

Наши четыре гибридных автомобиля помещены в различные классы конкретных компонентов, являющиеся подклассами класса `Auto`. Конструктор класса присваивает значение свойству `information`, а метод `price()` возвращает числовое значение цены автомобиля. В примерах с 4.39 по 4.42 в заголовках указаны имена файлов, в которых следует сохранить эти классы.

Пример 4.39. Prius.as

```
package
{
    public class Prius extends Auto
    {
        public function Prius()
        {
            information = "Toyota Prius Hybrid~\n";
        }
        override public function price():Number
        {
            return 21725.00;
        }
    }
}
```

Пример 4.40. Mariner.as

```
package
{
    public class Mariner extends Auto
    {
        public function Mariner()
        {
            information = "Mercury Mariner Hybrid~\n";
        }
        override public function price():Number
        {
            return 29225.00;
        }
    }
}
```

Пример 4.41. Accord.as

```
package
{
    public class Accord extends Auto
    {
        public function Accord()
        {
            information = "Accord Hybrid~\n";
        }
        override public function price():Number
        {
            return 30990.00;
        }
    }
}
```

Пример 4.42. Escape.as

```
package
{
    public class Escape extends Auto
    {
        public function Escape()
        {
            information = "Ford Escape Hybrid\n~";
        }
        override public function price():Number
        {
            return 26240.00;
        }
    }
}
```

Цены на автомобили, указанные в примерах, основаны на данных, найденных нами в Интернет, и они могут не совпадать с ценами в день прочтения вами нашей книги. Если вы хотите обновить цены или даже применить другие модели автомобилей, не стесняйтесь сделать это.

Специальные символы \n и тильда (~) используются в данных примерах для форматирования. Символ \n задает окончание строки с переходом на новую строку, а тильда (~) помогает отделить название модели от суммарной большой строки, получаемой после работы декораторов. Без них на выходе нашего приложения была бы неразбериха.

Использование опций автомобилей в качестве декораторов

Если все конкретные компоненты на месте, следующим нашим шагом будет конструирование класса декоратора и классов конкретных декораторов. Как и компонентам, которые они декорируют, им потребуются свойства название и цена, а также методы-получатели, которые будут их возвращать. Абстрактный декоратор должен быть таким, чтобы все производные от него классы имели в себе все необходимые свойства и методы.

Абстрактный декоратор для автомобильных опций

Как вы видели на примерах других абстрактных декораторов в этой главе, данный класс является одним из простейших. Поскольку он расширяет класс абстрактного компонента, он наследует все его свойства. Однако нам необходимо в нем заново имплементировать функцию-получатель, чтобы мы могли замещать ее в дальнейшем в различных конкретных декораторах. Пример 4.43 показывает нам класс абстрактного декоратора, который нужно сохранить в файле *Decorator.as*.

Пример 4.43. Decorator.as

```
package
{
    // Абстрактный класс
    public class Decorator extends Auto
    {
        override public function getInformation():String
        {
            return information;
        }
    }
}
```

Поскольку переменная information, представляющая собой абстрактную строку, наследуется нами из класса Auto, нам не нужно определять ее здесь заново.

Конкретные декораторы автомобильных опций

Все конкретные декораторы автомобильных опций генерируют некоторую информацию, которая добавляется к свойству `information` и к цене. По мере того как конкретный компонент упаковывается в конкретные декораторы, их строковые данные добавляются ко всем уже накопленным строчкам в компоненте. Поэтому, когда будет запущен метод `getInformation()`, он получит всю переданную информацию от всех опций и конкретного компонента, к которому они были применены. Чтобы при этом не получить на выходе громадную строку, которую будет невозможно разобрать, мы добавляем на конце строчек символ тильда (~), помогающий отделять друг от друга различные декорации. Примеры с 4.44 по 4.47 в своих заголовках содержат имена файлов, в которых их следует сохранить.

Пример 4.44. HeatedSeats.as

```
package
{
    public class HeatedSeats extends Decorator
    {
        private var auto:Auto;
        public function HeatedSeats(auto:Auto)
        {
            this.auto=auto;
        }
        override public function getInformation():String
        {
            return auto.getInformation() + " Heated Seats~";
        }
        override public function price():Number
        {
            return 350.78 + auto.price();
        }
    }
}
```

Пример 4.45. GPS.as

```
package
{
    public class GPS extends Decorator
    {
        private var auto:Auto;
        public function GPS(auto:Auto)
        {
            this.auto=auto;
        }
        override public function getInformation():String
        {
            return auto.getInformation() +
```

```
                " Global Positioning System~";
        }
        override public function price():Number
        {
            return 345.88 + auto.price();
        }
    }
}
```

Пример 4.46. RearViewVideo.as

```
package
{
    public class RearViewVideo extends Decorator
    {
        private var auto:Auto;
        public function RearViewVideo(auto:Auto)
        {
            this.auto=auto;
        }
        override public function getInformation():String
        {
            return auto.getInformation() + " Rear View Video~";
        }
        override public function price():Number
        {
            return 560.75 + auto.price();
        }
    }
}
```

Пример 4.47. Mp3.as

```
package
{
    public class MP3 extends Decorator
    {
        private var auto:Auto;
        public function MP3(auto:Auto)
        {
            this.auto=auto;
        }
        override public function getInformation():String
        {
            return auto.getInformation() + " MP3 Player~";
        }
        override public function price():Number
        {
            return 267.55 + auto.price();
        }
    }
}
```

Для данного конкретного приложения мы не занимаемся вопросами разделения цены на отдельные ее составляющие. Фактически нам здесь необходимо, чтобы все наши цены суммировались, в том числе и цена исходного конкретного компонента. Поэтому, хотя мы ставим в демаркационных целях на конце всех строчек знак тильда (~), в числовых значениях нам подобного разделения не нужно.

Конфигурация пользовательского интерфейса

Самый большой класс, который мы здесь будем использовать, предназначен для создания пользовательского интерфейса. Мы снабдили листинг данного класса в примере 4.48 номерами строчек, чтобы у нас была возможность впоследствии ссылаться на них. Этот класс по имени `Deal` («Сделка») выполняет работу по установке объектов интерфейса для пользователя. Они включают в себя переключаемые кнопки, отмечаемые позиции и обычные кнопки. Кроме того, немалый кусок кода требуется для вывода текстовых полей и их форматирования. Поэтому то, что интерфейс к шаблону Декоратор выглядит так громоздко, вовсе не вина этого шаблона проектирования. Фактически только приватные функции `getCar()` (начинается на строке 115) и `getOptions()` (начинается на строке 146) как-то используют информацию, получаемую из шаблона Декоратор.

Чтобы разобраться, что делает данный класс, введите код из примера 4.48 и сохраните его в файле *Deal.as*.

Пример 4.48. Deal.as

```
 1 package
 2 {
 3    import fl.controls.CheckBox;
 4    import fl.controls.RadioButton;
 5    import fl.controls.Button;
 6    import flash.display.Sprite;
 7    import flash.display.MovieClip;
 8    import flash.events.MouseEvent;
 9    import flash.text.TextField;
10    import flash.text.TextFormat;
11
12    public class Deal extends Sprite
13    {
14        internal var checks:Array=[];
15        internal var cars:Array=[];
16        internal var carDeal:Auto;
17        public var dealText:TextField=new TextField();
18
19        // Конструктор
20        public function Deal ():void
21        {
22            getRadios ();
```

```
23            getChecks ();
24            doDealButton ();
25            showDeal ();
26        }
27        // Добавить кнопку из библиотеки
28        private function doDealButton ():void
29        {
30            var doDeal:Button=new Button();
31            this.addChild (doDeal);
32            doDeal.x=215;
33            doDeal.y=195;
34            doDeal.label="Make Deal";
35            doDeal.addEventListener (MouseEvent.CLICK,getPackage);
36        }
37        //**
38        // Взять информацию из Декоратора и отобразить ее
39        //**
40        private function getPackage (e:MouseEvent):void
41        {
42            getCar ();
43            getOptions ();
44            if (carDeal == null)
45            {
46                return;
47            }
48            else
49            {
50              var nowDrive:String=carDeal.getInformation()+"\nTotal=$"+carDeal.price();
51            }
52            dealText.text=formatMachine(nowDrive);
53        }
54        // Форматировать информацию для вывода
55        private function formatMachine (format:String):String
56        {
57            if (format.indexOf("~") != -1)
58            {
59                format=format.split("~").join("\n");
60            }
61            return format;
62        }
63        // Текстовое поле и формат
64        private function showDeal ():void
65        {
66            dealText.width=150;
67            dealText.height=100;[1]
68            dealText.wordWrap=true;
69            dealText.multiline=true;
```

[1] **Значение** `dealText.height` **стоит поставить не менее 130, иначе при выборе более двух опций в текстовом поле не хватит места для вывода всей информации.** – ***Прим. науч. ред.***

```
            dealText.x=165;
            dealText.y=230;
            dealText.border=true;
            dealText.borderColor=0xccOOOO;
            var dealerFormat:TextFormat=new TextFormat();
            dealerFormat.leftMargin=4;
            dealerFormat.rightMargin=4;
            dealText.defaultTextFormat=dealerFormat;
            this.addChild (dealText);
        }
        // Добавить флаговую кнопку (checkbox) для опций (конкретных декораторов)
        private function getChecks ():void
        {
           var gizmos:Array=new Array("MP3","Heated Seats","GPS",  "Rear View Video");
            var saloon:uint=gizmos.length;
            var giz:uint;
            for (giz=0; giz<saloon; giz++)
            {
                checks[giz]=new CheckBox();
                this.addChild (checks[giz]);
                checks[giz].width=150;
                checks[giz].x=250;
                checks[giz].y=80+(giz*30);
                checks[giz].label=gizmos[giz];
            }
        }
        // Добавить переключатель автомобиля (конкретные компоненты)
        private function getRadios ():void
        {
            var car:Array=new Array("Escape","Mariner","Prius","Accord");
            var saloon:uint=car.length;
            var ride:uint;
            for (ride=0; ride<saloon; ride++)
            {
                cars[ride]=new RadioButton();
                cars[ride].groupName="deals";
                this.addChild (cars[ride]);
                cars[ride].x=150;
                cars[ride].y=80+(ride*30);
                cars[ride].label=car[ride];
            }
        }
        // Выбрать автомобиль и создать конкретный компонент
        private function getCar ():void
        {
            var tracker:String;
            var hybrid:uint;
            for (hybrid=0; hybrid<cars.length; hybrid++)
            {
                if (cars[hybrid].selected)
                {
```

```
                    tracker=cars[hybrid].label;
                    switch (tracker)
                    {
                        case “Escape” :
                            carDeal = new Escape();
                            break;

                        case “Mariner” :
                            carDeal = new Mariner();
                            break;

                        case “Prius” :
                            carDeal = new Prius();
                            break;

                        case “Accord” :
                            carDeal = new Accord();
                            break;
                    }
                }
            }
        }
        // Установить опции – упаковать конкретный компонент в декораторы
        private function getOptions ():void
        {
            var tracker:String;
            var toy:uint;
            for (toy=0; toy<checks.length; toy++)
            {
                if (checks[toy].selected)
                {
                    tracker=checks[toy].label;
                    switch (tracker)
                    {
                        case “MP3” :
                            carDeal = new MP3(carDeal);
                            break;

                        case “Heated Seats” :
                            carDeal = new HeatedSeats(carDeal);
                            break;

                        case “GPS” :
                            carDeal = new GPS(carDeal);
                            break;

                        case “Rear View Video” :
                            carDeal = new RearViewVideo(carDeal);
                    }
                }
            }
```

```
172          }
173     }
174 }
175
```

Создание документа Flash и конфигурация сцены

После того как вы сохраните файл *Deal.as*, вам нужно будет немного поработать с файлом документа Flash. Следуйте инструкциям, приведенным ниже.

1. Откройте новый документ Flash и сохраните его под именем *AutoDealer.fla* в том же каталоге, где содержатся остальные файлы *.as*.
2. В поле класс документа введите `Deal` и сохраните файл еще раз.
3. Используя рисунок 4.8 как образец, поместите статический текст «2 Guys From Connecticut Hybrid Cars» («Гибридные машины от двух парней из Коннектикута») в качестве заголовка. Под заголовком на горизонтальной позиции 150 введите `Select Car` («Выбрать Машину»), а на горизонтальной позиции 250 в той же строке введите `Select Options` («Выбрать Опции»).
4. Откройте панели `Library` и `Components` и перетащите копию переключающихся (RadioButton) и отмечаемых (CheckBox) кнопок в `Library`.
5. Выберите в меню Insert → Symbol и выберите в качестве его типа `Button` (обычная кнопка). В поле `Name` введите `Button` (что является именем по умолчанию) и отметьте позицию `Import for ActionScript`. Кликните на ОК, чтобы перейти в редактор символа `Button`, что будет означено появлением четверной временной линии.
6. В редакторе символа `Button` добавьте еще один уровень и выберите самый верхний уровень для редактирования. С помощью инструмента `Rectangle` («Прямоугольник») нарисуйте прямоугольник, цветом на ваш выбор, с закруглением углов в 8 пикселов, расположением на позиции x=0, y=0, размерами **W=43 (ширина) и H=13 (высота)**. Добавьте ключевые кадры для всех фреймов верхнего уровня. Выберите объект на фрейме `Over` и поменяйте (наоборот) цвет очертаний и цвет заливки.
7. Добавьте ключевые кадры первым трем фреймам нижнего уровня. Во фрейме `Up`, используя статический текст и шрифт «Narrow sans serif» высотой 10 пикселов, введите `Calculate` («Вычислить»), отцентрировав его прямо под кнопкой. Во фрейме `Over` сделайте то же самое, но введите `Total Cost` («Общая сумма»). Оставьте фрейм `Down` пустым.
8. Проверьте, что на вашей панели `Library` присутствуют следующие элементы: `CheckBox`, `CheckBoxSkins`, `Component Assets`, `Button`, `RadioButton`, `RadioButtonSkins`[1]. Если все это есть, то вы готовы к продолжению.

[1] `CheckBoxSkins` и `RadioButtonSkins` находятся в папке `Component Assets`. — *Прим. науч. ред.*

Теперь протестируйте ваше приложение. Все кнопки и текстовые поля должны автоматически разместиться на сцене. Рисунок 4.8 показывает, что вы можете ожидать увидеть, если у вас все работает, как планировалось.

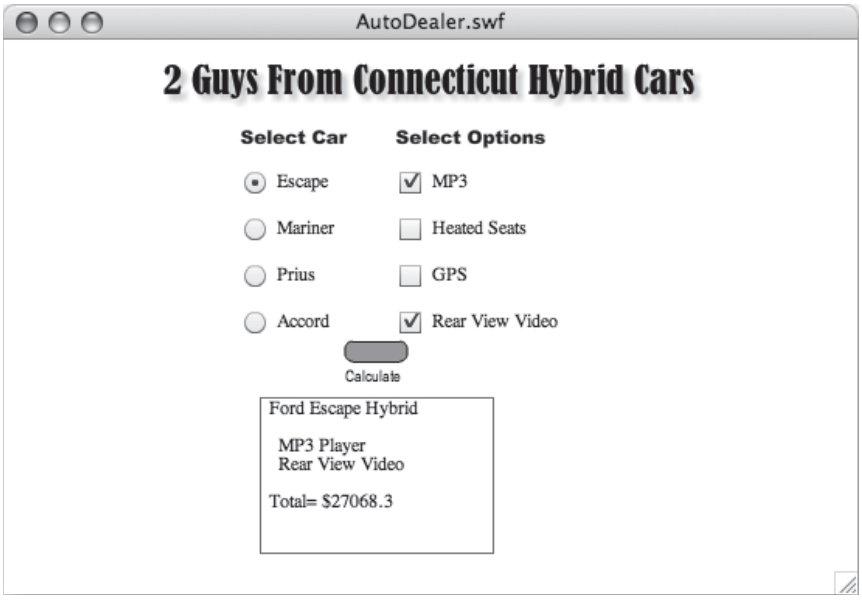


Рис. 4.8. *Интерфейс пользователя для Декоратора и результат его работы*

Имплементирование конкретных компонентов и их декораторов

Процесс имплементирования шаблона требует, чтобы определенные вещи были организованы так, чтобы пользователь мог выбрать, что именно ему нужно из конкретных компонентов (моделей машин) и из конкретных декораторов (опций) для них. Для начала наш скрипт должен определить, какая выбрана машина. Поэтому функция `getCar()` опрашивает в цикле переключаемые кнопки, пока не определит, какая из них выбрана. После этого она просто присваивает переменной `carDeal` объект конкретного компонента примерно следующим образом:

```
carDeal = new Mariner();
```

Объект `carDeal` должен находиться вне этой приватной функции, чтобы им можно было пользоваться не только в ней. В строке 16 примера 4.48 этот объект создается как представитель класса `Auto`. (Ключевое слово `internal` было добавлено для привлечения вашего внимания к тому факту, что данная переменная доступна всем элементам из данного пакета, несмотря на то, что эта характеристика присваивается по умолчанию при создании переменных.)

После присвоения объекта `carDeal` скрипт ищет выбранные опции. Для этого функция `getOptions()` проверяет у каждой отмечаемой кнопки, не была ли она выбрана. И если это оказывается именно так, то опция, то есть ее конкретный декоратор, упаковывает в себя конкретный объект `carDeal`. Например, если одним из вариантов выбора являются сидения с подогревом, то соответствующий условный фрагмент кода выглядит следующим образом:

```
case "Heated Seats":
carDeal = new HeatedSeats(carDeal);
break;
```

Поскольку может быть выбрано более одного декоратора, то конкретный объект может быть упакован несколько раз по мере нахождения отмеченных опций. Таким образом, любое количество опций может быть добавлено для выбранного автомобиля.

Заключение

Шаблон проектирования Декоратор превосходно добавляет новые характеристики своим базовым объектам без их фундаментальной переделки. Аналогичным образом украшения на новогодней елке полностью изменяют ее вид, не меняя при этом саму елку. Все, от внешнего вида веб-сайта до содержимого тележки для покупок в интернет-магазине, может быть изменено с использованием шаблона Декоратор.

Применение шаблона Декоратор следует рассматривать в первую очередь в тех случаях, когда ваш проект должен быть обновлен с добавлением новых объектов и новых характеристик к уже существующим. Нам нравится думать о Декораторе как о «шаблоне столетия». Если вообразить себе конкретные компоненты как типы людей, а их декорации как вид одежды, которую они носят, то не так уж сложно представить себе непрерывные обновления веб-сайта как изменение жизни людей за одно столетие – от кузнецов в начале XX века до нанотехнологов в начале XXI. Все типы людей могут быть представлены стилем их рабочей одеждой на данном отрезке времени, от кожаных фартуков кузнецов в 1900-м до скафандров космонавтов в 2000-м. Программист, думающий наперед на целое столетие, может с успехом использовать шаблон Декоратор в своих приложениях, и ему не потребуется менять свои базовые компоненты. Он просто добавит больше конкретных компонентов и декораторов к своему приложению.

Шаблон Декоратор не только позволяет добавлять новые характеристики без изменения структуры программы, но и дает возможность удалять лишние компоненты и декораторы, когда они станут вам не нужны. Более того, в нем очень просто проводить изменения в существующих компонентах и декораторах, не затрагивающие ничего, кроме их специфических строковых или числовых значений переменных.

Однако в шаблоне Декоратор имеются и слабые стороны. Например, при изучении данной главы у многих наших читателей могла возникнуть мысль, что они могли бы запрограммировать точно такую же функциональность для некоторых наших примеров, используя в несколько раз меньше кода и меньшее число классов. Это абсолютно верно, для некоторых приложений применение шаблона проектирования Декоратор подобно стрельбе по воробьям из пушки. Но, как и в случае других шаблонов проектирования, только разработчику выбирать, какой шаблон нужно применять в его программе, если применять вообще. Хотя это решение нельзя принимать, основываясь только на текущем размере программы. В начале любого проекта, в котором рассматривается применение шаблона Декоратор, вы должны спросить себя: будут мои конкретные компоненты и декораторы прибавляться и изменяться или все они будут довольно статичными? Даже если ваше исходное приложение выглядит как маленький желудь, вы должны понимать, что оно может стать могучим дубом, и строить соответствующие планы.

5

Шаблон Адаптер

Адаптируйся или погибни!
Теперь, как никогда ранее,
этот закон природы неумолим.

Герберт Уэллс

Представления и концепции человека
об окружающем мире формируются
не из чистого желания знания как такового,
а в борьбе и прогрессивной адаптации
себя к условиям жизни.

Эрнст Мах

Что такое шаблон Адаптер?

Основной целью использования шаблона Адаптер является расширение применения уже существующих классов. Например, предположим, что у нас имеется некоторый класс, который выполняет определенную работу и выполняет ее хорошо. Мы хотели бы использовать его в нашем приложении, но он не в полной мере удовлетворяет всем его текущим требованиям. Нам желательно было бы расширить его возможности или скомбинировать их с некоторыми другими классами для реализации дополнительной функциональности. Суть в том, что мы должны *адаптировать* существующий класс под новые требования. Именно это и делает шаблон проектирования Адаптер. Он позволяет клиенту использовать существующий класс благодаря конверсии его *интерфейса* в подходящий под новые условия. Главное – помнить, что при этом исходный класс не изменяется, а *класс адаптера*, имеющий нужный нам новый интерфейс, использует его или расширяет для обеспечения необходимой функциональности.

Шаблон проектирования для приучения к туалету

Хорошим примером адаптера является *туалетное сиденье* для малышей, устанавливаемое поверх стандартного сиденья туалета. Рассмотрим контекст этой ситуации. У нас есть *унаследованный объект* – сиденье туалета, основы дизайна которого не менялись уже много лет. Оно довольно неплохо функционирует в отношении своих взрослых пользователей (вот почему, наверно, и не менялся его дизайн). Теперь давайте взглянем на новый контекст ситуации, изображенный на рис. 5.1. Нам нужно адаптировать сиденье к использованию малышом. Проблема очевидна – *несовместимость интерфейсов*! Имеющееся сиденье туалета было сконструировано под взрослого, требуется конвертировать его интерфейс так, чтобы он подходил для маленького ребенка. Нам нужен адаптер, представляющий интерфейс, который соответствовал бы новому контексту. Детское туалетное сиденье было разработано именно для этого. С ним мы можем использовать существующий объект, интерфейс которого был адаптирован под ожидания клиента.

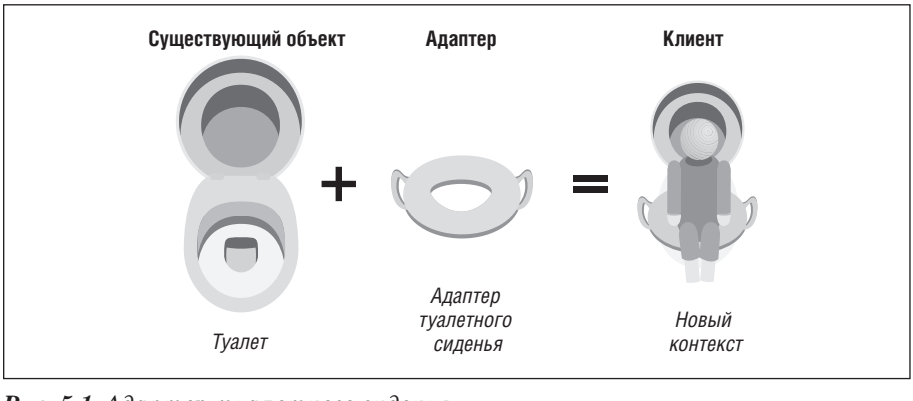


Рис. 5.1. *Адаптер туалетного сиденья*

В чем тут аналогия с шаблоном проектирования Адаптер? У нас есть существующий объект (сиденье туалета); новый интерфейс, которому он должен удовлетворять (попка малыша); адаптер, который конвертирует наш интерфейс (детское сиденье для туалета), и новый клиент (малыш).

Давайте расширим данную аналогию на код программы. Существующий объект будет тогда аналогом уже существующего класса. Новый интерфейс будет представлять собой новые требования к этому классу. А адаптер будет другим классом, имплементирующим новый интерфейс, как-то используя существующий класс.

Какова альтернатива использованию адаптера в данном случае? Мы можем полностью переписать или модифицировать код существующего класса, чтобы он удовлетворял новым требованиям. Это будет подобно покупке отдельного маленького туалета, разработанного специально для малышей (не подключаемого к канализации). Я полагаю, что боль-

шинство людей предпочтет все-таки не тратить на него дополнительные деньги и место в доме. Ведь это может быть так же неприятно, как изменение существующего, прекрасно работающего кода, сопряженное с появлением ошибок и нарушением зависимостей. В обеих ситуациях лучше применить адаптер.

Ключевые характеристики шаблона Адаптер

Чтобы вы могли переходить к рассмотрению вопроса об использовании шаблона Адаптер в вашем проекте, в последнем должна выполняться пара условий:

- Должен существовать класс, удовлетворяющий части требований к новой реализации, но имеющий интерфейс, не совместимый с новым контекстом его использования.
- Должен быть известен новый интерфейс для нового контекста.

Если оба условия выполняются, тогда мы можем разработать адаптер класса, который будет использовать существующий класс для имплементации нового *интерфейса*. Но перед планированием такой имплементации новый контекст использования класса должен быть тщательно проанализирован, а все новые требования – четко определены. Другими словами, мы должны все узнать о новом интерфейсе еще до того, как мы начнем рассматривать функциональность в существующих классах с целью выяснения, не подойдут ли они нам. Затем уже мы можем разрабатывать класс адаптера, имплементирующий желаемый интерфейс. Однако нет никакого смысла адаптировать существующий класс, если это не даст никаких преимуществ, например, экономии времени на разработку по сравнению с созданием совершенно нового класса.

Адаптеры объектов и адаптеры классов

Адаптер может задействовать существующий класс двумя способами. Он может использовать либо *композицию*, либо *наследование* для обеспечения доступа к свойствам и возможностям существующего класса. Если адаптер станет подклассом имеющегося класса, то он просто унаследует все его свойства и методы. Если адаптер будет применять композицию для доступа к существующему классу, то он будет содержать ссылку на его представителя и использовать его свойства и методы для имплементации. В ООП композиция в общем случае предпочтительнее наследования. Однако, как мы увидим в последующих примерах, есть веские причины для использования обоих подходов в зависимости от конкретного контекста. В любом случае идея состоит в том, чтобы использовать свойства и методы существующего класса для реализации новой требуемой функциональности.

Адаптеры объектов

Если адаптер использует *композицию* для получения доступа к существующему классу, его называют *адаптером объекта*. Такие адаптеры хранят в себе ссылку на представителя существующего класса. Следовательно, адаптер объекта *использует* этот класс.

Все классы адаптеров имплементируют какой-то *интерфейс*. В диаграмме классов, изображенной на рис. 5.2, интерфейс с именем `ITarget` декларирует единственный метод, называемый `request()`. На ней также показано, что ссылка на представителя класса `Adaptee` хранится в переменной `adaptee`. При реализации метода `request()` адаптер использует этого представителя.

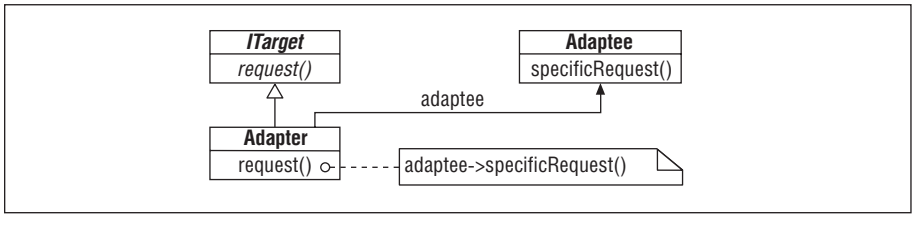


***Рис. 5.2.** Диаграмма классов для адаптера объекта*

Минималистский пример адаптера объектов

В нижеследующих примерах с 5.1 по 5.4 показана имплементация адаптера объекта на ActionScript 3.0. Имена для их файлов следующие: *Adaptee.as* (существующий класс), *ITarget.as* (требуемый интерфейс), *Adapter.as* (сам адаптер) и *Main.as* (клиент, а также *класс документа* Flash).

Пример 5.1. Adaptee.as

```
package
{
    public class Adaptee
    {
        public function specificRequest():void
        {
            trace("Called Adaptee:specificRequest()");
        }
    }
}
```

Пример 5.2. ITarget.as

```
package
{
public interface ITarget
```

```
    {
        function request():void
    }
}
```

Пример 5.3. Adapter.as

```
package
{
    public class Adapter implements ITarget
    {
        private var adaptee:Adaptee;

        public function Adapter()
        {
            this.adaptee = new Adaptee();
        }

        public function request():void
        {
            adaptee.specificRequest();
        }
    }
}
```

Пример 5.4. Main.as

```
package
{
    import flash.display.MovieClip;

    /**
    *    класс Main
    *    @ назначение: Класс документа
    */
    public class Main extends MovieClip
    {
        public function Main()
        {
            var target:ITarget = new Adapter();
            target.request();
        }
    }
}
```

Класс клиента `Main` создает представителя класса адаптера и вызывает метод `request()`, определенный в интерфейсе `ITarget`. Клиент видит только этот интерфейс, реализованный в адаптере. Адаптеры объектов подобны инспекторам дорожного движения, они останавливают все обращения к методам и решают, что с ними делать дальше. Некоторые из таких вызовов направляются напрямую к подходящим методам в исходном классе `Adaptee`. Класс `Adapter` жестко контролирует такой доступ

к существующему классу, полностью возлагая на себя имплементацию интерфейса ITarget. Клиент даже не знает, что адаптер при этом использует существующий класс. В классе Adaptee может быть еще много публично доступных методов и свойств, которые не доступны через адаптер.

Поскольку класс Adapter создает представителя класса Adaptee, между ними существует *жесткая связь*. Она не позволяет адаптеру использовать подклассы Adaptee или использовать совершенно другой существующий класс без модификации собственного кода.

Использование класса адаптера с параметром

Одним из вариантов адаптера из предыдущего примера может быть решение, при котором от клиента требуется передача представителя существующего класса при создании адаптера. Реализовать данное решение можно с помощью конструктора класса адаптера с параметром. Для этого в нем нужно произвести следующие изменения:

```
public function Adapter(a:Adaptee)
{
    this.adaptee = a;
}
```

Такая запись желательна для большинства случаев; благодаря ей снижается зависимость между классами Adapter и Adaptee, что обеспечивает большую гибкость реализации. Однако при этом на клиента накладываются обязательства по созданию представителя класса Adaptee и передаче его адаптеру:

```
var adaptee:Adaptee = new Adaptee();
var target:ITarget = new Adapter(adaptee);
```

Отрицательной стороной подобного решения является то, что теперь существующий класс становится видимым для клиента. Клиент создает представителя класса Adaptee, который передается классу Adapter. Это потенциально может привести к ошибкам в программе в случае, если клиент случайно начнет манипулировать данным представителем класса Adaptee.

Адаптеры классов

Если адаптер использует *наследование* для получения доступа к свойствам и методам существующего класса, то его называют *адаптером класса*. При этом класс адаптера расширяет существующий класс и имеет с ним отношение «является». Расширение существующего класса позволяет адаптеру просто унаследовать все его методы и свойства.

Как видно из диаграммы классов, имплементация адаптера класса требует *множественного наследования*. Эта концепция ООП позволяет классу быть наследником более чем одного суперкласса. На рис. 5.3 класс Adapter является наследником и класса Target, и класса Adaptee.

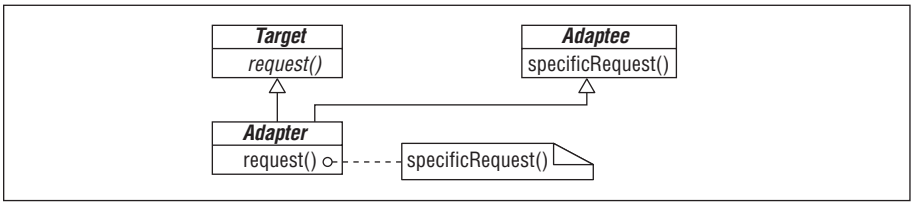


Рис. 5.3. *Диаграмма классов для адаптера класса*

Согласно классическому учебнику по шаблонам проектирования, написанному группой авторов, уважительно называемой «бандой четырех», одна из ветвей наследования класса адаптера предназначена для наследования *интерфейса*, а другая – для наследования *имплементации*. Данные авторы также указывали, что в большинстве случаев класс адаптера наследует интерфейс *публично*, в то время как наследование имплементации происходит *приватно*. Это скрывает ветвь наследования существующей имплементации, что приводит к желаемой ситуации, когда только публичный интерфейс остается видимым для клиентов. Например, на рис. 5.3 класс `Adapter` публично наследует класс `Target` и приватно класс `Adaptee`. В результате класс `Adapter` будет подклассом класса `Target` и будет использовать методы и свойства `Adaptee` для имплементации требуемых операций.

Реализация *множественного наследования* в языке программирования – непростая задача, и ActionScript 3.0 ее не поддерживает. Поэтому классическая имплементация адаптера с помощью ActionScript 3.0 невозможна. Однако классы в ActionScript 3.0 могут одновременно быть подклассами и имплементировать некоторый интерфейс. Если требуемый нам интерфейс называется `ITarget`, то мы можем определить класс адаптера следующим образом:

```
public class Adapter extends Adaptee implements ITarget
```

Здесь класс `Adapter` имплементирует интерфейс и одновременно с этим наследует класс `Adaptee`. Важно отметить, что данное наследование класса `Adaptee` происходит публично, что дает доступ к его функциональности. И хотя это не совсем чистая реализация класса адаптера, она может быть с пользой применена в определенных контекстах.

Минималистский пример адаптера класса

Примеры с 5.5 по 5.8 показывают имплементацию адаптера класса на ActionScript 3.0. Имена их файлов следующие: *Adaptee.as* (существующий класс), *ITarget.as* (требуемый интерфейс), *Adapter.as* (сам адаптер) и *Main.as* (клиент).

Пример 5.5. Adaptee.as

```
package
{
    public class Adaptee
```

```
    {
        public function requestA():void
        {
            trace("Called Adaptee:requestA()");
        }

        public function requestB():void
        {
            trace("Called Adaptee:requestB()");
        }

        public function requestC():void
        {
            trace("Called Adaptee:requestC()");
        }
    }
}
```

Пример 5.6. ITarget.as

```
package
{
    public interface ITarget
    {
        function renamedRequestA():void
        function requestB():void
        function requestC():void
        function requestD():void
    }
}
```

Пример 5.7. Adapter.as

```
package
{
    public class Adapter extends Adaptee implements ITarget
    {
        public function renamedRequestA():void
        {
            this.requestA();
        }

        override public function requestB():void
        {
            trace("Called Adapter:requestB()");
        }

        public function requestD():void
        {
            trace("Called Adapter:requestD()");
        }
    }
}
```

Пример 5.8. Main.as

```
package
{
    import flash.display.MovieClip;

    /**
    *    класс Main
    *    @ назначение: Класс документа
    */
    public class Main extends MovieClip
    {
        public function Main()
        {
            var target:ITarget = new Adapter();

            target.renamedRequestA();
            target.requestB();
            target.requestC();
            target.requestD();
        }
    }
}
```

Наш клиент (`Main`) будет генерировать следующий результат с помощью своих выражений `trace()`, показывающих какие класс и метод были задействованы.

```
Called Adaptee:requestA()
Called Adapter:requestB()
Called Adaptee:requestC()
Called Adapter:requestD()
```

Определение метода `renamedRequestA()` представляет собой простое переименование метода, который уже имплементирован в классе `Adaptee`. Это изменение осуществляется путем перенаправления запроса к `requestA()` в `Adaptee`.

Метод `requestB()` доступен из исходного класса `Adaptee`, но класс `Adapter` заместил его и имплементировал заново потому, что его функциональность не соответствовала требованиям.

Метод `requestC()` также доступен из класса `Adaptee`, и вызов метода передается напрямую ему. В этом состоит основное преимущество использования наследования: вам не нужно имплементировать метод заново, если вам нужны функциональность и *сигнатура метода*, имплементированная в исходном классе `Adaptee`.

Метод `requestD()` имплементирован в классе `Adapter`. Этому классу не всегда требуется использовать класс `Adaptee` для имплементации свойств и методов. Адаптер может иметь собственные свойства и методы для выполнения того, что ему нужно. Поэтому в интерфейсе `ITarget` могут со-

держаться методы, которые необходимы в новом контексте, но не имеют никакого отношения к существующему классу.

Ключевые концепции ООП в шаблоне Адаптер

Сравнение преимуществ и недостатков адаптеров классов и адаптеров объектов позволяет нам подробно разобрать тему предпочтения *композиции* перед *наследованием* в объектно-ориентированном проектировании программ.

Сравнение адаптеров классов и адаптеров объектов

Выбор в пользу адаптеров класса или адаптеров объекта сильно зависит от того контекста, в котором планируется их применять. Адаптеры объектов, использующие параметризованные конструкторы, являются достаточно гибкими, потому что объект исходного класса `Adaptee` передается им извне. Связь между классами `Adapter` и `Adaptee` разрушается, что позволяет адаптеру использовать представителей подкласса `Adaptee`.

В свою очередь адаптеры классов обычно требуют меньше кода, что позволяет быстрее их реализовывать. Поскольку они являются подклассом адаптируемого класса `Adaptee`, они находятся в отношении «являются» с этим классом. То есть класс `Adapter` того же типа, что и `Adaptee`, и с ним легче работать с позиции клиента. Клиенту для использования класса `Adapter` необходимо создать только его представителя, а не представителей обоих классов `Adapter` и `Adaptee` (как в случае применения параметризованного адаптера объекта). Если целевой интерфейс велик, адаптеры классов могут требовать значительно меньше времени на свою реализацию, поскольку большинство методов будут унаследованы.

Окончательное решение, что выбрать для имплементации, адаптер объекта или адаптер класса, зависит от множества факторов: требуемой гибкости реализации, размера целевого интерфейса, простоты реализации, степени совпадения между целевым интерфейсом и имеющимся классом.

В качестве общего правила запомните, что *композиция* предпочтительнее *наследования* из-за большей своей гибкости и меньшей связанности. Однако пример шаблона Адаптер показывает нам, что существуют случаи, когда вы можете сэкономить свое время и повысить эффективность своего кода с помощью *наследования*.

Пример: адаптер рулевого колеса

Шаблон Адаптер обычно используется, когда унаследованный вами существующий класс полностью обеспечивает нужную вам функциональность, но его интерфейс не удовлетворяет вашим требованиям. Представим, что у нас имеется унаследованный класс `LegacyCar`, являю-

щийся классом спрайта автомобиля из старой игры. Этот класс был изначально разработан для использования вместе с компьютерным рулевым колесом с целью обеспечения пользователю ощущения вождения, близкого к реальному. Мы изменим интерфейс класса `LegacyCar` так, чтобы нашей машиной можно было рулить, используя и другие компьютерные устройства ввода, такие как клавиатура и мышь.

Существующий класс

Класс `LegacyCar`, представленный в примере 5.9, является подклассом класса `Sprite`, и его надо сохранить в файле *LegacyCar.as*.

Пример 5.9. LegacyCar.as

```
package {

    import flash.display.*;
    import flash.events.*;
    import flash.geom.*;

    public class LegacyCar extends Sprite
    {

        internal var nSpeed:Number; //  скорость автомобиля в пикселах в кадр
        internal var nSteeringWheelAngle:Number; //  поворот руля в градусах

        public function LegacyCar(xLoc:int, yLoc:int)
        {
            nSpeed = 5;
            nSteeringWheelAngle = 0;
            this.drawCar();
            this.setLoc(xLoc, yLoc);
        }

        private function drawCar():void
        {
            //  рисуем корпус автомобиля
            graphics.beginFill(0x00FF00); //  зеленый цвет
            graphics.drawRect(-20, -10, 40, 20);
            graphics.endFill();
            //  рисуем шины
            drawTire(-12, -15);
            drawTire(12, -15);
            drawTire(-12, 15);
            drawTire(12, 15);
        }

        private function drawTire(xLoc:int, yLoc:int)
        {
            graphics.beginFill(0x000000); //  черный цвет
            graphics.drawRect(xLoc - 4, yLoc - 2, 8, 4);
```

```
38              graphics.endFill();
39          }
40
41
42          //  метод, устанавливающий позицию x и y спрайта
43          private function setLoc(xLoc:int, yLoc:int):void
44          {
45              this.x = xLoc;
46              this.y = yLoc;
47          }
48
49          //  метод, добавляющий обработчики событий и запускающий автомобиль
50          public function start():void
51          {
52              if (this.stage)
53              {
54                  //  добавляем обработчик события ENTER_FRAME -  doMoveCar()
55                  this.addEventListener(Event.ENTER_FRAME, this.doMoveCar);
56              } else {
57                  throw new Error("Add car to display list first")
58              }
59          }
60
61          //  метод, устанавливающий угол поворота руля (в градусах)
62          public function setSteeringWheelAngle(nAngle:int):void
63          {
64              nSteeringWheelAngle = nAngle;
65          }
66
67          //  перемещаем автомобиль
68          private function doMoveCar(event:Event):void
69          {
70              this.rotation += nSteeringWheelAngle * 0.01; // поворачиваем спрайт
71              trace(nSteeringWheelAngle);
72              var newLocOffset:Point = Point.polar(nSpeed,
73                                      this.rotation * Math.PI / 180);
74              this.x += newLocOffset.x; // перемещаем по оси x
75              this.y += newLocOffset.y; // перемещаем по оси y
76              // устанавливаем спрайт в центр сцены, если он уехал за край экрана
77              if ((this.y < 0) || (this.y > this.stage.stageHeight) ||
78                  ((this.x < 0)) || (this.x > this.stage.stageWidth))
79              {
80                  this.setLoc(this.stage.stageWidth * 0.5,
81                                      this.stage.stageHeight * 0.5);
82              }
83          }
84      }
85  }
```

Параметризованный *конструктор* класса (строки 13–19) получает начальное положение нашей машины через параметры `xLoc` и `yLoc`. Кон-

структор устанавливает свойство nSpeed («скорость») в его значение по умолчанию 5, а свойство nSteeringWheelAngle («угол поворота рулевого колеса») – в ноль, что означает, что машина едет прямо. Затем он вызывает метод drawCar(), чтобы нарисовать машину, и метод setLoc() с координатами x и y, чтобы расположить ее спрайт.

Метод drawCar() (строки 21–32) использует свойство graphics класса Sprite, чтобы изобразить машину, как она выглядит сзади сверху (так называемый «взгляд на карту»). Машина представляет собой зеленый прямоугольник с четырьмя черными колесами (кажущимися сверху тоже прямоугольными), которые чертятся с помощью метода drawTire() (строки 34–39). Машина рисуется таким образом, что ее *опорная точка* находится в центре спрайта.

Класс LegacyCar имеет два публичных метода. Первый метод start() (строки 50–59) контролирует, был ли спрайт автомобиля добавлен к *списку отображения*, путем проверки существования объекта Stage (спрайт может быть добавлен к списку отображения с помощью метода addChild()). Если объект Stage существует, то данный метод регистрирует подписку метода doMoveCar() («продвинуть машину») на события *enter frame* («начало нового кадра») (строка 55). Метод doMoveCar() продвигает машину вперед на nSpeed пикселов в каждом кадре. Данный класс LegacyCar не предназначен для контроля за скоростью машины или остановки ее движения.

Второй публичный метод (строки 62–65) – setSteeringWheelAngle(); он принимает угол поворота рулевого колеса в градусах в качестве своего параметра и устанавливает его как значение свойства nSteeringWheelAngle. Свойство nSteeringWheelAngle представляет собой текущий угол поворота рулевого колеса нашего автомобиля.

Метод doMoveCar() (строки 68–83) является обработчиком события *enter frame* (подписка на получение которого была для него зарегистрирована в строке 55). Такое событие генерируется всякий раз, когда при проигрывании Flash начинается новый кадр. Частота кадров может быть установлена внутри документа Flash. При каждом новом событии *enter frame* выполняется метод doMoveCar(). Он поворачивает машину на 1 процент угла поворота рулевого колеса (хранящегося в nSteeringWheelAngle). Например, если текущий угол поворота рулевого колеса отрицателен и составляет –200 градусов, то спрайт автомобиля будет поворачиваться на 2 градуса против часовой стрелки при каждом новом событии *enter frame*. Данный метод затем использует функцию polar() класса Point, чтобы определить следующее положение спрайта нашей машины, основываясь на ее скорости (nSpeed) и текущем направлении движения (свойство rotation класса Sprite). После чего спрайт передвигается на это новое место. Последней задачей doMoveCar() является проверка, не вылетела ли наша машина за пределы экрана. Если вылетела, то она просто возвращается в центр сцены.

Конечно, класс LegacyCar не самая функционально полная машина, которую мы только видели, однако он позволяет надежно управлять автомобилем с помощью компьютерного рулевого колеса. И мы надеемся использовать эту его возможность в новом контексте, в котором необходимо управление с помощью других устройств ввода.

Конверсия интерфейса

Основной функцией шаблона Адаптер является конверсия интерфейса существующего класса под требования нового контекста. Чтобы выяснить ее применимость, мы сначала должны понять существующий интерфейс, а затем определить требования нового контекста. После этого мы сможем решить, подходит ли конверсия интерфейса существующего класса с использованием шаблона Адаптер для конкретной задачи.

Исходный интерфейс

Класс LegacyCar имеет два публичных метода, которые и составляют его интерфейс. Публичные методы являются традиционным способом, с помощью которого клиент может получить доступ к объекту и манипулировать им. Публичный метод start() просто начинает движение нашей машины. Он работает подобно ключу зажигания автомобиля (в предположении, что ее передача всегда включена). Следующий публичный метод setSteeringWheelAngle() получает на входе угол поворота рулевого колеса и соответствующим образом поворачивает автомобиль. Данный интерфейс класса проиллюстрирован на рис. 5.4.

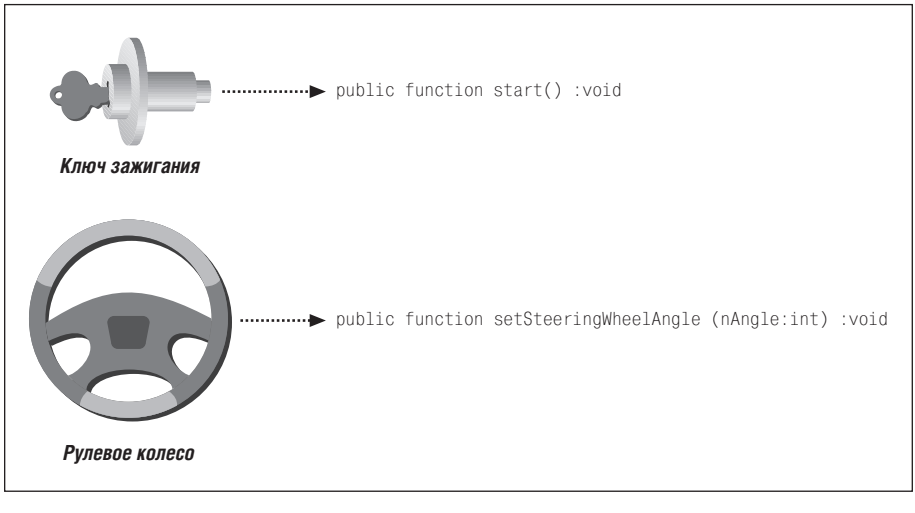


Рис. 5.4. *Исходный интерфейс класса LegacyCar*

Новый контекст

Будем полагать, что новый контекст игры требует, чтобы машина управлялась не только с помощью компьютерного рулевого колеса, но и другими устройствами ввода. Например, нам нужно уметь управлять ею с помощью клавиатуры и мыши. Представьте, что новый контекст связан с необходимостью модифицировать машину так, чтобы ею мог управлять тот, кто не может использовать для этого рулевое колесо. Исходный интерфейс позволял очень точно поворачивать машину с помощью метода `setSteeringWheelAngle()`. Рулевое колесо можно было повернуть на полградуса или на 900 градусов (два с половиной оборота руля). Однако генерирование такого широкого диапазона значений с помощью нажатия клавиш невыполнимо. Одна клавиша клавиатуры по существу имеет только два значения: она либо нажата, либо нет. Очевидно, нам придется поворачивать постепенно, на некоторые дискретные значения угла. Например, мы можем приравнять каждое нажатие клавиши со стрелкой влево к повороту рулевого колеса на 50 градусов против часовой стрелки. А клавиша со стрелкой вправо будет поворачивать руль по часовой стрелке. Кроме того, будет весьма полезно иметь возможность возвращать рулевое колесо в исходное состояние. Для этого можно использовать клавишу со стрелкой вверх, которая будет устанавливать угол поворота в ноль. И нам нужно конвертировать наш существующий интерфейс к данному новому контексту.

Новый интерфейс

Основываясь на нашем предыдущем анализе ситуации, мы можем сделать вывод, что новому интерфейсу необходимо иметь несколько методов, чтобы удовлетворять требованиям нового контекста. Наш целевой интерфейс `ICar`, разработанный для нового контекста, приведен в примере 5.10 и проиллюстрирован на рис. 5.5.

Пример 5.10. ICar.as

```
package
{
    public interface ICar
    {
        function start():void
        function turnMoreToTheLeft():void
        function turnMoreToTheRight():void
        function goStraight():void
    }
}
```

Класс адаптера

Теперь у нас есть требуемый интерфейс. Следующий наш шаг – выбор между адаптером класса и адаптером объекта. Поскольку существующий класс весьма мал и имеет только два публичных интерфейса, ис-

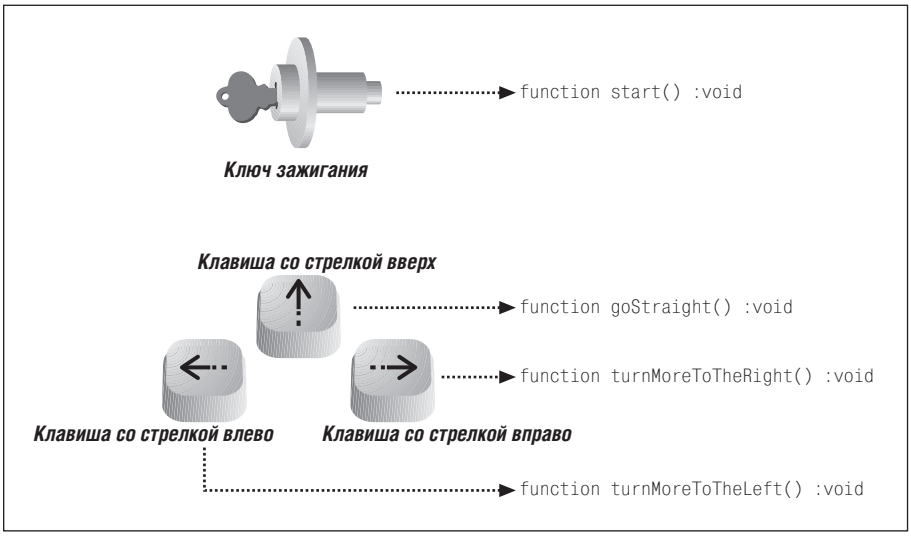


Рис. 5.5. *Интерфейс поворота с помощью клавиатуры*

пользование адаптера класса не даст большого выигрыша. Если бы в существующем классе были методы, которые можно было бы использовать напрямую в новом контексте, тогда использование адаптера класса имело бы смысл. Следовательно, в нашем классе `CarAdapter` мы будем имплементировать адаптер объекта, что и показано в примере 5.11.

Пример 5.11. CarAdapter.as

```
package
{
    public class CarAdapter implements ICar
    {

        private var legacyCar:LegacyCar;
        private var nSteeringAngle:Number;

        public function CarAdapter(car:LegacyCar)
        {
            this.legacyCar = car;
            this.nSteeringAngle = 0;
            this.goStraight();
        }

        public function start():void
        {
            legacyCar.start();
        }

        public function turnMoreToTheLeft():void
```

```
        {
            legacyCar.setSteeringWheelAngle(nSteeringAngle -= 50);
        }

        public function turnMoreToTheRight():void
        {
            legacyCar.setSteeringWheelAngle(nSteeringAngle += 50);
        }

        public function goStraight():void
        {
            legacyCar.setSteeringWheelAngle(nSteeringAngle = 0);
        }

    }
}
```

Класс `CarAdapter` имплементирует интерфейс `ICar` и имеет параметризованный конструктор, который получает себе на вход представителя класса `LegacyCar`. Кроме того, данный класс имеет четыре публичных метода, как описано в интерфейсе. Все четыре метода используют один и тот же публичный метод класса `LegacyCar` в своих имплементациях. Угол поворота руля меняется дискретными шагами в 50 градусов, как мы предложили ранее. Обратите внимание на новое свойство `nSteeringAngle`, которое хранит в себе текущий угол поворота руля автомобиля. Класс адаптера может иметь свои собственные свойства и методы, необходимые для его имплементации.

Клиент

Нам осталось только создать клиента для тестирования нашего адаптера. Наш клиент (*Main.as* из примера 5.12) создает нового представителя класса `LegacyCar` и передает два параметра его конструктору, которые помещают наш автомобиль в центр сцены. Далее он создает представителя класса `CarAdapter` и передает ему ранее созданного представителя `LegacyCar`. Чтобы поворачивать автомобиль, нашему клиенту надо получать ввод с клавиатуры. Для этого клиент устанавливает подписку на события *key_down* для своей функции `onKeyPress()`. Обработчик данного события вызывает соответствующие методы класса адаптера в зависимости от полученного ввода с клавиатуры. Функция-обработчик `onKeyPress()` реагирует на нажатие стрелок влево, вправо и вверх в событиях *key_down* и вызывает соответственно методы `turnMoreToTheLeft()`, `turnMoreToTheRight()` и `goStraight()`, имплементированные в классе адаптера.

Пример 5.12. Main.as (класс документа)

```
package
{
    import flash.display.MovieClip;
```

```
        import flash.text.*;
        import flash.events.*;
        import flash.ui.*;

        public class Main extends MovieClip
        {
            private var carAdapter:ICar; //  объявляем переменную адаптера

            public function Main()
            {
                //  создаем реализацию автомобиля
                var legacyCar:LegacyCar =
                    new LegacyCar(this.stage.stageWidth * 0.5,
                                  this.stage.stageHeight * 0.75);
                addChild(legacyCar); //  добавляем автомобиль в список отображения

                //  упаковываем автомобиль в CarAdapter
                this.carAdapter = new CarAdapter(legacyCar);

                //  добавляем на сцену обработчик события KEY_DOWN onKeyPress()
                this.stage.addEventListener(KeyboardEvent.KEY_DOWN,
                    this.onKeyPress);

                carAdapter.start();
            }

            private function onKeyPress(event:KeyboardEvent):void
            {
                switch (event.keyCode)
                {
                    case Keyboard.LEFT :
                        this.carAdapter.turnMoreToTheLeft();
                        break;
                    case Keyboard.RIGHT :
                        this.carAdapter.turnMoreToTheRight();
                        break;
                    case Keyboard.UP :
                        this.carAdapter.goStraight();
                        break;
                }
            }
        }
    }
```

Обратите внимание, новый интерфейс поворачивает автомобиль дискретными шагами (по 50 градусов на каждое нажатие клавиши со стрелками). Следовательно, чтобы сделать резкий поворот направо, нужно нажать и удерживать некоторое время клавишу со стрелкой вправо (чтобы сгенерировать множественное повторение события *key_down*). Одиночное нажатие соответствующих клавиш дает только малые углы поворота.

Расширенный пример: поворот машины с помощью мыши

Если целевой интерфейс проектировался с учетом требований гибкости и дальнейшего его применения, то полученный адаптер можно будет использовать во многих контекстах. Поскольку наше рулевое управление с поворотом на дискретные углы реализовано в нашем адаптере с применением изолированных методов `turnMoreToTheLeft()`, `turnMoreToTheRight()` и `goStraight()`, его можно использовать и для управления с помощью других устройств ввода, таких как мышь или джойстик. Давайте рассмотрим, как клиент может применять компьютерную мышь для поворота автомобиля. Например, мы можем использовать щелчок кнопкой мыши как указание, что руль нужно вернуть в исходное положение, а автомобилю – ехать прямо. Но как нам реализовать поворот налево или направо с помощью мыши? Одним из решений является реагирование на горизонтальное движение мышью, что показано на рис. 5.6. Если мышь двигается налево, мы можем вызвать метод `turnMoreToTheLeft()`, а если направо – метод `turnMoreToTheRight()`.

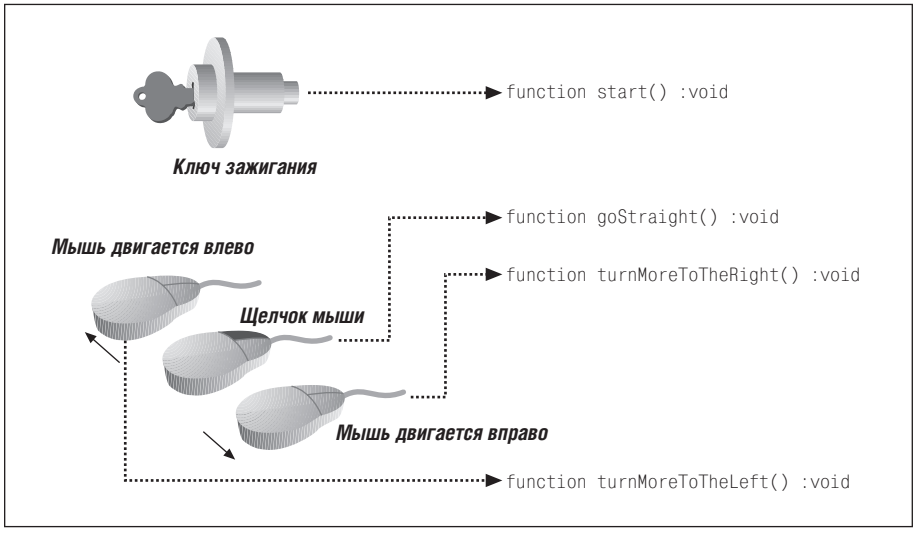


***Рис. 5.6.** Интерфейс управления мышью*

Наш новый клиент для данной ситуации очень похож на класс `Main` (пример 5.12), в котором для поворота руля использовался ввод с клавиатуры. Чтобы для управления машиной можно было использовать компьютерную мышь, в классе `Main` нужно произвести следующие изменения. Нам потребуется новое свойство, хранящее предыдущую горизонтальную позицию мыши на сцене, чтобы можно было понять, направо или налево она движется. Исходной позицией мыши будет ноль.

```
private var oldMouseStageX:Number = 0;
```

Другим изменением будет подписка различных функций обработчиков на события, связанные с мышью, а не с клавиатурой. Вместо того чтобы подписать обработчика на событие *key down*, нам придется подписать обработчиков на события *mouse down* и *mouse move*. Обратите внимание на то, что эти обработчики событий следует прикрепить к сцене, а не к объекту LegacyCar. Это позволит им реагировать на щелчки мышью по всей области сцены, а не только по изображению автомобиля.

```
this.stage.addEventListener(MouseEvent.MOUSE_MOVE, this.doMouseMove);
this.stage.addEventListener(MouseEvent.MOUSE_DOWN, this.doMouseDown);
```

Теперь нам нужно реализовать сами функции-обработчики выбранных событий мыши, вызывающие соответствующие методы в адаптере. Функция doMouseDown() будет обрабатывать события *mouse down* и вызывать при этом метод нашего адаптера goStrait().

```
private function doMouseDown(event:MouseEvent):void
{
    this.carAdapter.goStraight();
}
```

Другая функция обработки событий doMouseMove() будет реагировать на события *mouse move*. Первым делом она проверит, в какую сторону двигается мышь, налево или направо от своей предыдущей позиции, хранящейся в свойстве oldMouseStageX. При обнаружении движения мыши налево или направо будут вызваны соответственно методы адаптера turnMoreToTheLeft или turnMoreToTheRight().

```
private function doMouseMove(event:MouseEvent):void
{
    //  изменение в горизонтальном положении мыши
    var xDiff = event.stageX - this.oldMouseStageX;
    if (xDiff > 0) {
        this.carAdapter.turnMoreToTheRight();
    } else {
        this.carAdapter.turnMoreToTheLeft();
    }
    event.updateAfterEvent(); //  обработать это событие в первую очередь
    this.oldMouseStageX = event.stageX //  сохранить старое положение мыши по
горизонтали
}
```

Таким образом, наш пример наглядно показывает, что хорошо продуманный адаптер может быть использован в различных ситуациях и разных контекстах.

Пример: адаптер для отображения списка

Шаблон проектирования Адаптер бывает полезен также в случаях, когда требуется создать модифицированные классы на основе стан-

дартных классов с целью реализации в них нужной функциональности. Представьте, например, что вы хотите отобразить список некоторых строковых значений на сцене. В нем могут содержаться различные вещи: список имен людей, список результатов в игре, список продуктов в тележке для покупок и т. д. Во Flash вы можете использовать представителя класса `TextField` для отображения списка, помещая каждую из позиций в его отдельную строчку. Однако это требует, чтобы ваш текст был заранее отформатирован и каждая позиция из списка для отображения отделялась в нем символом возврата каретки (‘`\r`’ или код ASCII 13). Только тогда объект `TextField` представит его корректно. Следующий фрагмент кода отображает список с помощью объекта класса `TextField`.

```
var listField = new TextField();
listField.x = 20;
listField.y = 20;

var sList:String = “Bread”;
sList += “\r” + “Butter”;
sList += “\r” + “Cheese”;

listField.text = sList;

addChild(listField);
```

Данный фрагмент, конечно, сработает, но что произойдет, если вы захотите удалить из вашего списка одну из позиций? Вам придется создавать список заново. Таким образом, хранение позиций исходного списка в массиве и запись его в строку с разделителями из символов возврата строки оказываются предпочтительнее. Это, правда, потребует больше кода, и наш маленький фрагмент программы будет только предтечей общего решения.

Но будет ли удобно пользоваться таким классом `ListDisplayField`, в который можно будет добавлять по одному элементу списка за раз? Он будет действовать подобно классу `TextField` с добавлением возможности работы со списками. Это хороший пример контекста для разработки адаптера, поскольку мы собираемся использовать существующий класс и конвертировать его интерфейс под новые задачи.

Существующий интерфейс

Класс `TextField` используется в ActionScript 3.0 для отображения и ввода текста. Он будет нашим исходным классом. В его интерфейсе содержится множество свойств и методов для работы на низком уровне с отображаемым текстом. В документации по ActionScript 3.0 (доступной онлайн) этот класс описан очень подробно.

Конверсия интерфейса

Нам нужно конвертировать интерфейс класса `TextField` и внести в него функции добавления и удаления элементов списка, чего требует новый контекст. Метод `deleteItemAt(n:uint)` будет возвращать признак успеха или провала операции по удалению элемента списка в указанной позиции. Кроме того, в нашем интерфейсе будет определена функция очистки всего поля. Наш новый целевой интерфейс `IListDisplay` приведен в примере 5.13.

Пример 5.13. IListDisplay.as

```
package
{
    public interface IListDisplay
    {
        function addItem(s:String):void;
        function deleteItemAt(n:uint):Boolean;
        function clear():void;
    }
}
```

Класс адаптера

До имплементации интерфейса `IListDisplay` нам необходимо решить, какой тип адаптера, класса или объекта, для этого использовать. Существующий интерфейс класса `TextField` очень насыщен и содержит несколько свойств и методов, которые, несомненно, будут для нас очень полезны и в новом контексте. Например, в интерфейсе `TextField` определено множество свойств, таких как `backgroundColor` и `border`, позволяющих осуществлять форматирование представления объекта этого класса. Кроме того, в нем определено несколько методов, таких как `setTextFormat()`, которые позволяют форматировать сам текст с использованием объекта `TextFormat`. Если мы будем имплементировать адаптер `ListDisplayField` как адаптер класса с использованием наследования, то мы сможем выиграть в эффективности реализации и размере кода. Однако при такой имплементации в ActionScript 3.0 все публичные методы и свойства `TextField` будут доступны клиентам. Нам придется аккуратно переписать несколько методов и свойств данного класса, позволяющих прямое манипулирование его текстом. Реализация нашего адаптера показана в примере 5.14.

Пример 5.14. ListDisplayField.as

```
1 package
2 {
3     import flash.text.*;
4
5     //  Адаптер
6     public class ListDisplayField extends TextField implements IListDisplay
```

```
 7    {
 8
 9        private var aList:Array;
10
11        public function ListDisplayField()
12        {
13            super(); //  Вызываем конструктор TextField
14            this.clear();
15        }
16
17        public function addItem(s:String):void
18        {
19            this.aList.push(s);
20            this.update();
21        }
22
23        public function deleteItemAt(i:uint):Boolean
24        {
25            if ((i > 0) && (i <= aList.length)) {
26                aList.splice(i-1, 1);
27                this.update();
28                return true;
29            } else {
30                return false;
31            }
32        }
33
34        public function clear():void
35        {
36            aList = [];
37            this.update();
38        }
39
40        internal function update()
41        {
42           var listText:String = "";
43           // обработаем массив для создания строки, разделенной переводами каретки
44           for (var i:Number = 0; i < aList.length; i++) {
45               listText += aList[i] + "\r";
46           }
47           super.text = listText;
48        }
49
50        override public function set text(s:String):void
51        {
52            throw new Error("Cannot directly set text property - use
                        addItem() method");
53        }
54
55        override public function set htmlText(s:String):void
56        {
```

```
57              throw new Error("Cannot directly set htmlText property");
58          }
59
60          override public function appendText(s:String):void
61          {
62              throw new Error("Cannot append text - use addItem() method");
63          }
64
65          override public function replaceSelectedText(s:String):void
66          {
67              throw new Error("Cannot replace selected text");
68          }
69
70          override public function replaceText(beginIndex:int, endIndex:int,
                                                 newText:String):void
71          {
72              throw new Error("Cannot replace text");
73          }
74      }
75  }
```

Наш класс адаптера `ListDisplayField` расширяет класс `TextField` и имплементирует интерфейс `IListDisplay`. В его конструкторе вызывается метод `super()`, который вызывает конструктор его суперкласса `TextField` (строка 13). Дальнейшая имплементация адаптера довольно наглядна, в том смысле, что все элементы отображаемого списка хранятся в массиве `aList` («список»), являющемся свойством данного класса. Метод `addItem(s:string)` добавляет переданную ему строку в конец массива `aList`. Метод `deleteItemAt(i:uint)` удаляет элемент списка в соответствующей позиции (нумерация элементов списка начинается с 1). Он также проверяет, находится ли указанная ему позиция внутри текущих границ массива, и возвращает соответствующее логическое значение. Обратите внимание на указатель `super`, использованный в методе `update()` (строка 47). Он позволяет данному методу напрямую обращаться к свойству суперкласса `text`. Это необходимо, поскольку метод-установщик для свойства `text` был замещен (строки 50–53). На самом деле несколько публичных методов в классе `TextField`, позволяющих прямое заполнение поля содержимым, было замещено, чтобы предотвратить возможное вмешательство со стороны клиента в функциональность, указанную в `IListDisplay`. Среди замещенных методов есть методы-установщики свойств `text` и `htmText`, включая `appendText()`, `replaceSelectedText()` и `replaceText()`.

Клиент

Наш клиент будет использовать адаптер `ListDisplayField` точно так же, как он использовал бы класс `TextField`. Это является большим преимуществом для имплементации именно адаптера класса на ActionScript 3.0, поскольку все публичные методы и свойства существующего класса

продолжают функционировать, как раньше (за исключением специально замещенных). Наш клиент *Main.as* из примера 5.15 использует объект TextFormat, чтобы установить шрифт и размер шрифта для отображения объекта. Кроме того, он использует свойство border класса TextField, чтобы отобразить границу текстового поля.

Пример 5.15. Main.as (класс документа)

```
package
{
    import flash.display.MovieClip;
    import flash.text.*;

    public class Main extends MovieClip
    {
        public function Main()
        {
            //  создаем ListDisplayField
            var shoppingListField:ListDisplayField = new ListDisplayField();

            //  задаем стиль текста
            var format:TextFormat = new TextFormat();
            format.size = 18;
            format.font = "Arial";

            //  устанавливаем стиль и местоположение текстового поля
            shoppingListField.x = 20;
            shoppingListField.y = 20;
            shoppingListField.border = true;
            shoppingListField.defaultTextFormat = format;

            //  создаем список
            shoppingListField.addItem("Bread");
            shoppingListField.addItem("Butter");
            shoppingListField.addItem("Broccoli");
            shoppingListField.addItem("Cheese");
            //  передумали насчет Broccoli
            shoppingListField.deleteItemAt(3);

            addChild(shoppingListField);
        }
    }
}
```

Данный код клиента продемонстрирует на сцене текстовое поле, расположенное в точке (20,20), со словами Bread, Butter и Cheese в трех разных строчках, выведенными шрифтом Arial размером 18. Теперь отображение списка слов и удаление его отдельного элемента в заданной позиции стали для клиента очень просты и удобны. Все эти возможности имплементированы в нашем адаптере и инкапсулированы в нем. Обрати-

те внимание на то, что дополнительные методы по манипулированию списком могут быть легко добавлены в наш адаптер. Например, метод удаления элементов по их содержанию, а не позиции или метод замены или вставки элемента списка может быть реализован путем расширения адаптера. Таким образом, мы создали универсальный класс, который можно использовать в различных контекстах путем конвертирования его интерфейса под новые требования.

Расширенный пример: вывод списка новых книг издательства O'Reilly

Чтобы продемонстрировать вам возможности использования нашего адаптера `ListDisplayField`, мы используем его для отображения списка новых книг, опубликованных в O'Reilly Media, Inc. Поскольку этот список часто меняется, имеет смысл получать его динамически, прямо из Интернета, и отображать список книг с помощью нашего адаптера.

Список новых книг издательства O'Reilly публикуется на его сайте в виде *новостной ленты*. Новостная лента – это формат данных, используемый для публикации часто меняющегося контента. Публикатор такого контента позволяет другим пользователям *подписываться* на данную новостную ленту.

Новостная лента издательства O'Reilly, посвященная его новым книгам (показанная в примере 5.16), публикуется с использованием формата *Atom Syndication Format*, известного как *Atom*. Другой общепринятый формат публикации новостных лент в Интернете – RSS (акроним RSS означает «Really Simple Syndication»). Все такие форматы задаются с помощью языка XML (Extensible Markup Language), который является средством для создания специальных языков разметки, описывающих данные.

Пример 5.16. Сжатая версия новостной ленты издательства O'Reilly, показывающая названия двух книг

```
<?xml version='1.0' encoding='utf-8'?>

<feed xmlns='http://www.w3.org/2005/Atom'
xmlns:itunes='http://www.itunes.com/dtds/podcast-1.0.dtd'
xml:lang='en-US'>

<title>O'Reilly Media, Inc. New Books</title>
<link rel="alternate" type="text/html" href=http://www.oreilly.com/
    hreflang="en" title="O'Reilly Media, Inc. New Books" />
<rights>Copyright O'Reilly Media, Inc.</rights>
<updated>2006-09-28T22:45:16-08:00</updated>
<link rel="self" type="application/atom+xml" href=
    "http://www.oreillynet.com/pub/feed/29"/>
```

```
<entry>
    <title>Designing and Building Enterprise DMZs</title>
    <id>http://www.oreilly.com/catalog/1597491004</id>
    <link rel='alternate' href='http://www.oreilly.com/
        catalog/1597491004'/>
    <summary type='html'>
    &lt;i>Building DMZs for Enterprise Networks&lt;/i> covers
        a sorely needed area in critical business infrastructure:
        the Demilitarized Zone. DMZs play a crucial role in any
        network consisting of a Hosted Internet Web Server, internal
        servers which need to be segregated, External to Internal DNS
        Server, and an E-mail SMTP Relay Agent. This book covers what
        an administrator needs to plan out and integrate a DMZ into
        a network for small, medium, and Enterprise networks.
    </summary>
    <author><name>Hal Flynn</name></author>
    <updated>2006-09-28T22:45:16-08:00</updated>
</entry>

<entry>
    <title>Windows PowerShell Quick Reference</title>
    <id>http://www.oreilly.com/catalog/windowspowershell</id>
    <link rel='alternate' href='http://www.oreilly.com/catalog/
        windowspowershell'/>
    <summary type='html'>
    For years, support for scripting and command-line administration
        on the Windows platform has paled in comparison to the
        support offered by the Unix platform. Unix administrators
        enjoyed the immense power and productivity of their command
        shells, while Windows administrators watched in envy.
        Windows PowerShell, Microsoft's next-generation command shell
        and scripting language, changes this landscape completely.
        This Short Cut contains the essential reference material
        to help you get your work done-including the scripting language
        syntax, a regular-expression reference, useful .NET classes,
        and much more.
    </summary>
    <author><name>Lee Holmes</name></author>
    <updated>2006-09-27T22:46:17-08:00</updated>
</entry>
</feed>
```

В нашей программе мы подпишемся на данную новостную ленту, выделим из нее названия книг, используя средства «ECMAScript for XML» (E4X), и выведем их на экран, используя наш адаптер `ListDisplayField`**. Язык ActionScript 3.0 полностью поддерживает E4X, позволяя манипулировать документом XML с помощью выражений, подражающих синтаксису этого языка. Пример 5.17 демонстрирует нашу имплементацию.**

Пример 5.17. Main.as (класс документа)

```
1 package
2 {
3     import flash.display.MovieClip;
4     import flash.text.*;
5     import flash.events.*;
6     import flash.net.*;
7
8     /**
9     *     класс Main
10    *     @ назначение: Класс документа
11    */
12    public class Main extends MovieClip
13    {
14        var xml:XML;
15        var xmlLoader:URLLoader;
16        var newBookListField:ListDisplayField;
17
18        public function Main()
19        {
20            //  создаем ListDisplayField (адаптер)
21            newBookListField = new ListDisplayField();
22
23            //  выбираем стиль текста
24            var format:TextFormat = new TextFormat();
25            format.size = 14;
26            format.font = "Arial";
27
28            //  устанавливаем стиль и положение текстового поля
29            newBookListField.x = 20;
30            newBookListField.y = 20;
31            newBookListField.width = 500;
32            newBookListField.height = 300;
33            newBookListField.border = true;
34            newBookListField.defaultTextFormat = format;
35
36            //  создаем список из O'Reilly New Books Feed (Atom)
37            var newBooksURL = "http://www.oreillynet.com/pub/feed/29";
38            xml = new XML();
39            var xmlURL:URLRequest = new URLRequest(newBooksURL);
40            xmlLoader = new URLLoader(xmlURL);
41            xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
42
43            addChild(newBookListField);
                          // добавляем текстовое поле в список отображения
44        }
45
46        function xmlLoaded(evtObj:Event)
47        {
48            xml = XML(xmlLoader.data);
49            //  устанавливаем пространство имен XML по умолчанию
```

```
50              if (xml.namespace("") != undefined)
51              {
52                  default xml namespace = xml.namespace("");
53              }
54              //  добавляем в ListDisplayField названия новых книг
55              for each (var bookTitle:XML in xml..entry.title)
56              {
57                  newBookListField.addItem(bookTitle.toString());
58              }
59          }
60      }
61  }
```

Интернет-адрес нашей новостной рассылки (*http://www.oreillynet.com/pub/feed/29*) указан в строке 37. Метод `URLLoader()` (строка 40) загружает ее документ из Интернета и вызывает функцию `xmlLoaded()`, когда загрузка завершена. Метод `xmlLoaded()` регистрируется как обработчик события завершения загрузки данного документа (строка 41), поскольку загрузка данных – это асинхронная операция, требующая по ее завершении специального уведомления. Важно отметить, что если в XML-документе устанавливается некоторое пространство имен, то в методе устанавливается пространство имен по умолчанию (строки 50–53). Это позволяет E4X правильно выполнять проверки при разборе XML-документа. Выражение `for each` получает название каждой книги (строки 55–58) и добавляет его в адаптер `ListDisplayField`. Рисунок 5.7 показывает пример результата работы нашей программы.

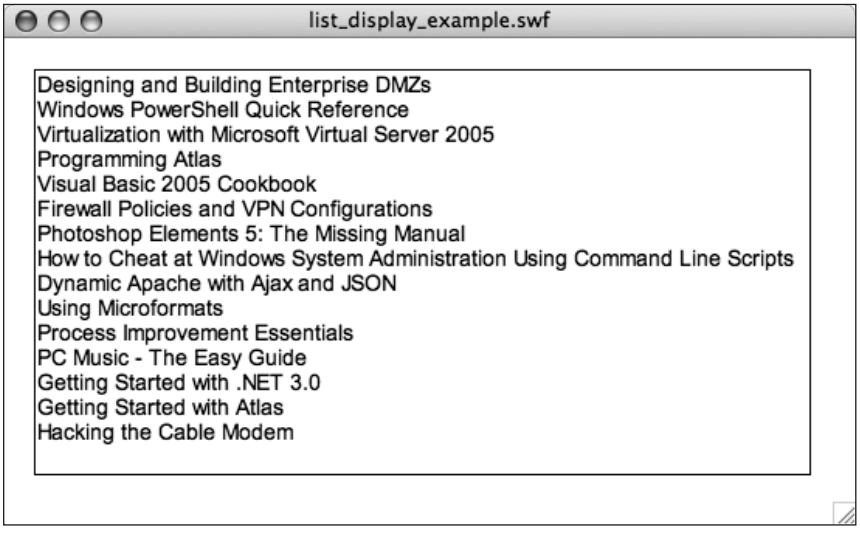


Рис. 5.7. *Список новых книг издательства O'Reilly, показанный с помощью адаптера ListDisplayField*

Заключение

Шаблон проектирования Адаптер позволяет применять имеющиеся классы в ситуациях, которых не существовало при их разработке или в которых не предполагалось их участие. Например, в приложении может присутствовать много унаследованного кода, который более никем не поддерживается, но должен быть использован по некоторым причинам. Классы адаптера используют или расширяют существующий код, чтобы он удовлетворял новым условиям. Главной функцией шаблона Адаптер является конверсия интерфейса существующего класса под требования нового контекста применения.

Шаблон Адаптер очень близок по своему замыслу и действию к некоторым другим шаблонам, таким как Фасад, Декоратор (см. главу 4) и Мост. Его главным отличием от них является то, что для реализации Адаптера необходимы целевой интерфейс, который он имплементирует, существующий класс, который Адаптер использует или расширяет для своей имплементации, и отсутствие требований по упрощению интерфейса.

Существует два типа реализации адаптера, и выбор между адаптером класса и адаптером объекта подчеркивает преимущества и недостатки композиции и наследования для имплементации. Эти типы иллюстрируют также важность интерфейсов и то, как разработчик может в них инкапсулировать все детали имплементации.

6

Шаблон Компоновщик

Чему нам всем нужно научиться – так это работать в системе, то есть я хочу сказать, что каждый из нас, каждая команда, каждая платформа, каждый отдел, каждый компонент существует не для собственной выгоды или признания, а для привнесения вклада в общее дело на взаимовыгодной основе.

У. Эдвардс Деминг

В логически совершенном языке для каждого простого объекта будет только одно слово, а все непростые вещи будут выражаться комбинацией слов, составленной, конечно же, из слов для простых вещей, по одному слову для каждого простого компонента.

Бертран Рассел

Сложная работоспособная система неизбежно оказывается развитием более простой работоспособной системы.

Джон Голь

Что такое шаблон Компоновщик?

Шаблон Компоновщик является надежным решением для построения сложных систем из нескольких более мелких компонентов. Компоненты, которые составляют такую систему, могут быть как отдельными объектами, так и контейнерами, содержащими в себе целую коллекцию объектов. Например, автомобиль представляет собой сложную систему, образованную из нескольких более мелких компонентов. Он включает в себя двигатель, корпус, подвеску, сиденья, колеса и т. д. Для упро-

щения описания давайте полагать колесо неделимой или примитивной частью автомобиля. Их в автомобиле четыре штуки. (На самом деле колесо само состоит из шины, колпака, обода и т. д.) В автомобиле есть еще и одно рулевое колесо. Двигатель тоже содержит в себе несколько меньших компонентов, таких как цилиндры, компрессор, радиатор и т. д. Двигатель является компонентом автомобиля, но при этом и он сам является коллекцией различных компонентов. То есть мы называем компонентом также составной объект, представляющий собой коллекцию других компонентов. Прелесть шаблона Компоновщик в том, что он позволяет клиентам рассматривать свои примитивные и составные объекты одним и тем же способом. Например, добавляя или удаляя в него компонент, клиенту совершенно не нужно думать, является ли данный объект примитивным или составным. Клиент может легко удалить и двигатель, и колесо через один общий интерфейс.

Для лучшего понимания шаблона Компоновщик полезно представлять себе сложный составной объект в виде некоторого иерархического дерева. Мы имеем в виду перевернутые деревья, как на рис. 6.1, где система начинается с корневого узла и спускается каскадами вниз, разделяясь на отдельные ветки.

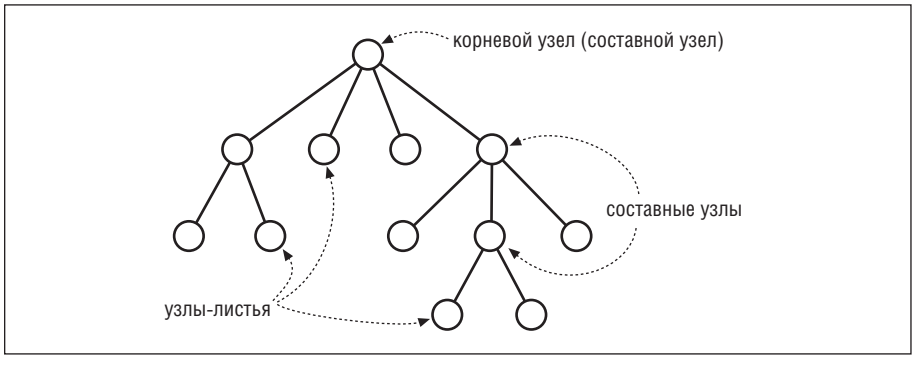


Рис. 6.1. *Структура иерархического дерева с показанными узлами и листьями*

На рис. 6.1 узлы, содержащие другие компоненты, – составные объекты. В качестве узлов-листьев (оконечных узлов) выступают неделимые, или примитивные, объекты, которые не могут иметь собственных детей. Каждый листовой узел является *дочерним узлом* составного узла, и у каждого составного узла может быть несколько детей, включая другие составные узлы. Составной узел, который возвышается в иерархии непосредственно над данным листом, называется его *родителем*. Как показано на диаграмме классов на рис. 6.2, шаблон Компоновщик обеспечивает общий интерфейс для работы с составными и листовыми узлами.

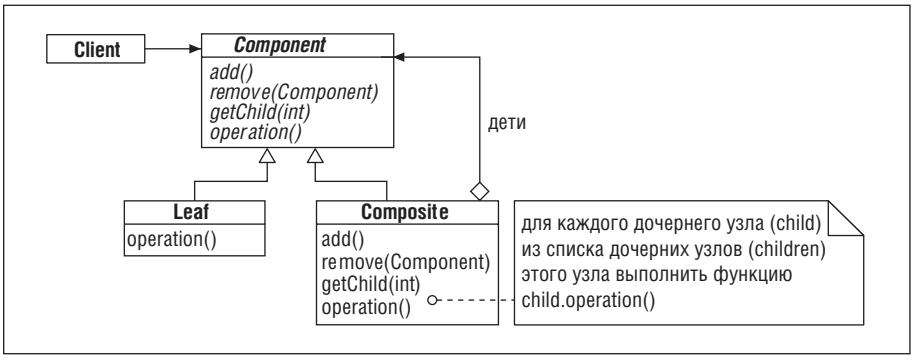


***Рис. 6.2.** Диаграмма классов шаблона Компоновщик*

Общим интерфейсом для составных и листовых узлов является класс `Component` («компонент»). Он обычно определяется как абстрактный класс, поэтому в нем может быть задана имплементация по умолчанию и для составных, и для листовых узлов. В классе `Component` должны быть объявлены как минимум следующие методы: `add()`, `remove(child: Component)` и `getChild(n:int)`. С их помощью клиент может построить свою составную систему. Классы `Leaf` («лист») и `Composite` («составной») расширяют класс `Component` и замещают в нем необходимые методы. Имплементация по умолчанию в классе `Component` обычно создается для конкретных классов `Leaf`. Поскольку добавление, удаление и доступ к детям не применимы для листовых узлов, в имплементации по умолчанию вызов этих методов `Component` приводит к возникновению исключения (ошибочной ситуации). Поэтому эти методы должны быть замещены и имплементированы в классе `Composite`. Обратите внимание, что составной объект может состоять из нескольких детей типа `Component`. Свойство `children` («дети») из класса `Composite` агрегирует в себе все компоненты детей данного класса.

Реальная мощь шаблона Компоновщик проявляется в том, как имплементирован метод `operation()` в классах `Leaf` и `Composite`, что показано на диаграмме на рис. 6.2. Операции, которые работают и в составных узлах, и в листьях, первоначально определяются в классе `Component`. В большинстве случаев такие операции определяются как абстрактные методы, что заставляет классы `Leaf` и `Composite` замещать их своими имплементациями. Составным классам приходится предоставлять рекурсивную имплементацию метода `operation()`. Это требует вызова метода `operation()` в каждом из его детей, ссылки на которых хранятся в свойстве `children`. Если говорить в терминах иерархического дерева, то вызов метода `operation()` обойдет дерево, вызывая метод `operation()` во всех дочерних компонентах.

Чтобы понять, как общие операции применяются рекурсивно ко всем своим детям, посмотрите на аналогию шаблона Компоновщик – подвижную детскую игрушку мобиль (рис. 6.3).

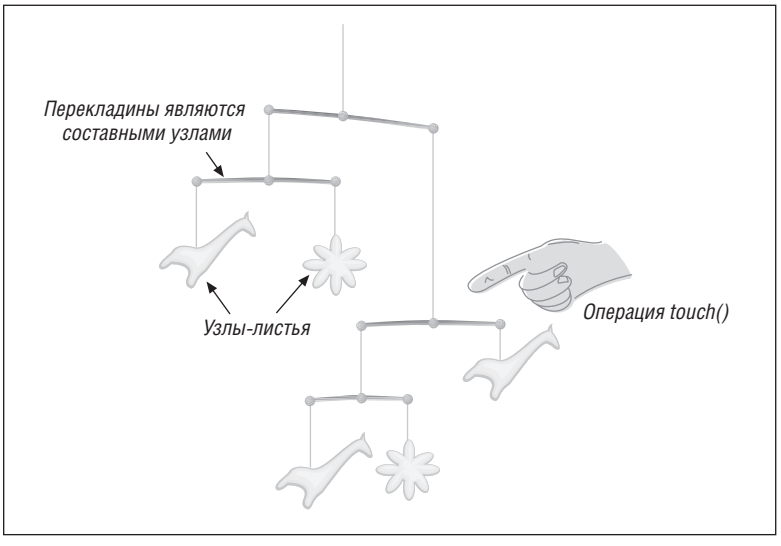


Рис. 6.3. *Игрушка мобиль имплементирует шаблон Компоновщик*

Перекладины в этой игрушке представляют собой составные узлы, а жирафики и звездочки – узлы-листья. Мобиль позволяет ясно представить, что произойдет, если вы тронете один из компонентов. Мы будем полагать, что операция `touch()` («дотронуться») применима в данном случае ко всем компонентам в мобиле. Прикосновение к любому из компонентов мобиля заставляет этот компонент вращаться. Но если вы дотронетесь до составного узла, то вращаться будет не только он, но и все его дети. А касание листового узла, такого как звездочка, будет вращать только его, поскольку у листьев нет дочерних компонентов. Тот факт, что клиенту, манипулирующему сложной структурой, не нужно волноваться, с чем именно он работает, с листом или составным узлом, является ключевой особенностью шаблона Компоновщик.

Ключевые особенности шаблона Компоновщик

Шаблон Компоновщик облегчает построение и манипулирование сложными структурами, состоящими из нескольких связанных элементов.

- Эти комплексные структуры строятся как иерархические деревья.
- Компоненты структуры могут быть одиночными (примитивными, или неделимыми, объектами) либо составными, содержащими наборы других компонентов.
- Структура позволяет клиентам рассматривать одиночные компоненты (листовые узлы) и составные компоненты (составные узлы) одним и тем же образом, что существенно упрощает ее интерфейс.

Минималистский пример шаблона Компоновщик

Данный пример имплементирует композицию, показанную на рис. 6.2. Файл *Component.as* (пример 6.1) содержит абстрактный класс `Component`, который определяет интерфейс как для листовых, так и для составных узлов. Файл *Leaf.as* (пример 6.2) содержит класс `Leaf`, а файл *Composite.as* (пример 6.3) – класс `Composite`. Оба эти класса являются расширением класса `Component` и обеспечивают для него все необходимые имплементации. Файл *Main.as* (пример 6.4) содержит класс клиента `Main`, являющийся также классом документа Flash.

Пример 6.1. Component.as

```
package
{
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class Component
    {
        public function add(c:Component):void
        {
            throw new IllegalOperationError
                ("add operation not supported");
        }

        public function remove(c:Component):void
        {
            throw new IllegalOperationError
                ("remove operation not supported");
        }

        public function getChild(n:int):Component
        {
            throw new IllegalOperationError
                ("getChild operation not supported");
            return null;
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        public function operation():void {}

    }
}
```

Класс `Component` должен рассматриваться как абстрактный класс, представители которого не должны создаваться в программе. Он определяет абстрактный интерфейс работы с дочерними компонентами и реализует имплементацию по умолчанию для методов `add(c:Component)`,

remove(c:Component) и getChild(n:int). Имплементация по умолчанию для данных методов разработана для узлов-листьев и вызывает исключительную ситуацию, создавая состояние ошибки IllegalOperationError («недопустимая операция»). Все это сделано для листовых узлов, потому что они не могут иметь дочерних компонентов и не должны реализовывать операции, которые работают с ними. Кроме того, метод operation() определяется как абстрактный метод без какой-либо имплементации; ее обеспечение оставлено на долю дочерних классов класса Component.

Пример 6.2. Leaf.as

```
package
{
    public class Leaf extends Component
    {
        private var sName:String;

        public function Leaf(sNodeName:String)
        {
            this.sName = sNodeName;
        }

        override public function operation():void
        {
            trace(this.sName);
        }
    }
}
```

Класс Leaf является расширением класса Component. В нем определяется свойство sName, которое будет содержать имя конкретного листа. Он также имплементирует параметризованный конструктор, который принимает строковое значение и устанавливает его в качестве указанного имени в свойстве sName. Кроме того, он имплементирует метод operation(), распечатывая имя из sName с помощью выражения trace(). В результате вызова метода operation() в листовом узле происходит вывод его имени на панель Output.

Пример 6.3. Composite.as

```
package
{
    public class Composite extends Component
    {
        private var sName:String;
        private var aChildren:Array;

        public function Composite(sNodeName:String)
        {
            this.sName = sNodeName;
```

```
            this.aChildren = new Array();
        }

        override public function add(c:Component):void
        {
            aChildren.push(c);
        }

        override public function operation():void
        {
            trace(this.sName);
            for each (var c:Component in aChildren)
            {
                c.operation();
            }
        }
    }
}
```

Класс Composite также расширяет класс Component и определяет в себе свойство sName, которое содержит в себе имя данного составного узла, передаваемое его параметризованному конструктору. Уникальная особенность класса Composite заключается в том, что в нем нужно определить структуру, которая будет содержать ссылки на всех его детей, и имплементировать все методы по работе с ними. Простейшей структурой, способной содержать ссылки на детей, является массив. Он называется aChildren и инициализируется в конструкторе. Какой бы компонент ни добавлялся композитному узлу с помощью метода add(c:Component), он будет добавляться в массив aChildren с помощью метода push(). Метод данного класса operation() также уникален, в том смысле, что он не только распечатывает имя sName, но и вызывает operation() во всех детях. Таким образом, метод operation() будет рекурсивно вызван во всех нижележащих дочерних узлах данной древовидной структуры.

Пример 6.4. Main.as

```
package
{
    import flash.display.MovieClip;

    public class Main extends MovieClip
    {
        public function Main()
        {
            var root:Composite = new Composite("root");

            //  создаем узел
            var n1:Composite = new Composite("composite 1");
            n1.add(new Leaf("leaf 1")); // добавляем в него лист
```

```
            n1.add(new Leaf("leaf 2")); // добавляем другой лист
            root.add(n1); //  добавляем этот узел в корневой

            // создаем другой узел
            var n2:Composite = new Composite("composite 2");
            n2.add(new Leaf("leaf 3")); // добавляем в него лист
            n2.add(new Leaf("leaf 4")); // еще один лист
            n2.add(new Leaf("leaf 5")); // и еще один лист
            root.add(n2); //  добавляем узел в корневой

            // добавляем лист в корневой узел
            root.add(new Leaf("leaf 6"));

            root.operation(); // вызываем operation в корневом узле
        }
    }
}
```

Класс `Main`, показанный в примере 6.4, представляет собой *клиента*. Он создает составной *корневой* узел и добавляет ему несколько составных и листовых узлов. Конечная составная структура показана графически на рис. 6.4.

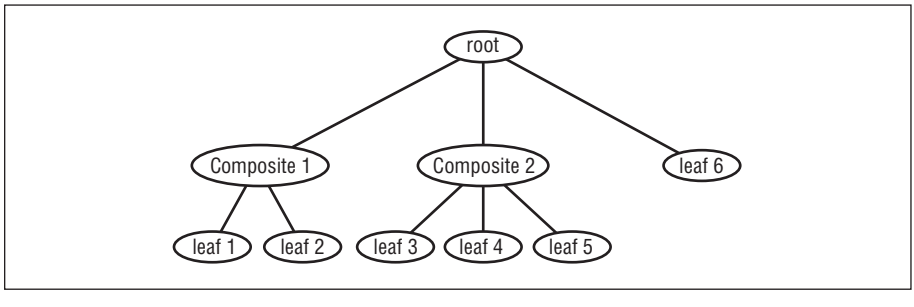


Рис. 6.4. *Составная древовидная структура, построенная клиентом*

В конце конструктор класса `Main` вызывает метод `operation()` корневого узла. Данный метод рекурсивно выводит имена всех нижележащих компонентов структуры. Ниже приводится результат работы этой операции.

```
root
composite 1
leaf 1
leaf 2
composite 2
leaf 3
leaf 4
leaf 5
leaf 6
```

Важно отметить, что *клиент* видит только интерфейс, определенный в классе `Component`. Ему не нужно различать составные и листовые узлы и знать, как именно в них реализован метод `operation()`. Он может просто вызвать этот метод у любого составного или листового узла и получить соответствующий результат.

Доступ к дочерним узлам

Как вы могли заметить, мы пока не стали замещать и имплементировать метод `getChild(n:int)` в классе `Composite`. Тем не менее данный метод очень важен в шаблоне Компоновщик, поскольку он позволяет клиентам создавать сложные составные структуры, не объявляя собственных переменных, как было сделано в предыдущем примере. Это гарантирует правильную работу сборщика мусора при удалении узлов. Сборщик мусора дает возможность приложению возвращать память и ресурсы, освободившиеся после удаления узлов. Мы обсудим его более подробно далее в этой главе. Метод `getChild()` позволяет клиенту получать доступ к дочерним узлам любого составного узла. Ниже приводится его имплементация в классе `Composite`.

```
override public function getChild(n:int):Component
{
    if ((n > 0) && (n <= aChildren.length))
    {
        return aChildren[n-1];
    } else {
        return null;
    }
}
```

Параметризованный метод `getChild()` возвращает дочерний объект под номером, указанным в его параметре `n` (нумерация начинается с 1). После проверки границ массива метод возвращает либо дочерний объект, либо `null`, если такого дочернего узла не существует. Теперь клиент может использовать метод `getChild()` и строить свои составные структуры более эффективно.

```
//  создаем корневой узел
var root:Composite = new Composite("root");

//  добавляем узел в корневой
root.add(new Composite("node 1"));
root.getChild(1).add(new Leaf("leaf 1"));
root.getChild(1).add(new Leaf("leaf 2"));

//  добавляем другой узел в корневой
root.add(new Composite("node 2"));
root.getChild(2).add(new Leaf("leaf 3"));
root.getChild(2).add(new Leaf("leaf 4"));
root.getChild(2).add(new Leaf("leaf 5"));
```

```
// добавляем лист в корневой узел
root.add(new Leaf("leaf 6"));

root.operation(); // вызываем operation в корневом узле
```

Удаление узлов

Имплементация метода `remove()` в классе `Composite` происходит немного сложнее. Чтобы надежно удалять узлы, нужно делать это со стороны их родительского узла. Поэтому, прежде чем имплементировать удаление узлов, нам нужно создать от каждого узла ссылку на его родителя.

Создание ссылки на родителя

Поскольку у всех узлов, кроме корневого, будут ссылки на родительский узел, имеет смысл создать ссылку на родителя и методы работы с ней прямо в общем классе `Component`. Мы можем определить свойство `parentNode`, хранящее ссылку на родителя, следующим образом:

```
protected var parentNode:Composite = null;
```

Обратите внимание, что типом данного свойства указан класс `Composite`, а значением по умолчанию – `null`. Это, конечно, приводит к определенной зависимости между классами `Component` и `Composite`, однако дает правильное указание типа нашей ссылки. Кроме того, наше свойство объявляется как *защищенное* (`protected`), что позволяет ограничить доступ к нему только текущим классом и его подклассами. Два метода для установки и получения данной ссылки на родителя должны быть также определены в классе `Component`.

```
internal function setParent(compositeNode:Composite):void
{
    this.parentNode = compositeNode;
}

public function getParent():Composite
{
    return this.parentNode;
}
```

Обратите внимание, что метод `getParent()` определен как *публичный* (`public`), чтобы он был доступным для клиентов. Однако метод `setParent()` определен как *внутренний* (`internal`), чтобы запретить установку ссылки на родителя извне данного *пакета*. Это важно, поскольку ссылка на родителя должна устанавливаться только в методе `add()`, имплементированном в классе `Composite`. Метод `add()` должен быть замещен следующим образом, чтобы устанавливать ссылку на родителя в дочернем классе на текущий составной узел.

```
override public function add(c:Component):void
{
    aChildren.push(c);
```

```
        c.setParent(this);
    }
```

Выполнение следующих выражений даст нам доступ к *третьему дочернему узлу второго дочернего узла корневого узла*, т. е. к *листу 5*. Его родителем будет второй дочерний узел корневого узла (см. рис. 6.4).

```
var leaf5:Component = root.getChild(2).getChild(3);
var leaf5Parent:Composite = leaf5.getParent();
```

Вызов метода `operation()` у родителя *листа 5* должен дать следующий результат:

```
composite 2
leaf 3
leaf 4
leaf 5
```

Имплементация удаления узлов

При удалении узлов очень важно полностью удалить все ссылки на удаляемые объекты, чтобы сборщик мусора мог возвратить использовавшуюся ими память. Например, перед удалением составного узла нужно удалить всех его детей. Если кто-то из них сам является составным узлом, то необходимо также рекурсивно удалить его детей. Следовательно, удаление будет работать сходным образом с методом `operation()`.

Из предыдущего сценария видно, что нам придется по-разному работать с листовыми и составными узлами при их удалении. Поэтому весьма полезно определить метод `getComposite()` в классе `Component`, который будет возвращать составной объект, если его объект действительно составной, и `null` – в противном случае. По умолчанию он будет возвращать `null`. Мы определим следующим образом имплементацию данного метода по умолчанию в классе `Component`.

```
internal function getComposite():Composite
{
    return null;
}
```

А в классе `Composite` мы заместим этот метод и будем просто возвращать данный составной объект.

```
override internal function getComposite():Composite
{
    return this;
}
```

Кроме того, ссылки на родителя у компонентов должны быть удалены до удаления самих компонентов. Для этого следующий метод удаления таких ссылок должен быть определен и имплементирован в классе `Component`.

```
internal function removeParentRef():void
```

```
{
    this.parentNode = null;
}
```

После этого мы можем, наконец, заместить метод `remove(c:Component)`, определенный и имплементированный по умолчанию в классе `Component`, его новой имплементацией в классе `Composite` (пример 6.5).

Пример 6.5. Метод remove()

```
 1 override public function remove(c:Component):void
 2 {
 3     if (c === this)
 4     {
 5         // удалить всех детей
 6         for (var i:int = 0; i < aChildren.length; i++)
 7         {
 8             safeRemove(aChildren[i]); // удалить детей
 9         }
10         this.aChildren = []; // удалить ссылки на детей
11         this.removeParentRef(); // удалить ссылку на родителя
12     } else {
13         for (var j:int = 0; j < aChildren.length; j++)
14         {
15             if (aChildren[j] == c)
16             {
17                 safeRemove(aChildren[j]); // удалить дочерний узел
18                 aChildren.splice(j, 1); // удалить ссылку на него
19             }
20         }
21     }
22 }
```

Метод `remove()` получает на входе один параметр, который является представителем класса `Component`. В зависимости от переданного ему компонента метод `remove()` должен разобраться с двумя различными ситуациями: (а) что делать, если удаляемый компонент является текущим объектом, и (б) что делать, если это не так. Если переданный компонент представляет собой текущий объект, то нужно рекурсивно удалить все его дочерние компоненты (строки 6–9), а затем удалить его ссылку на родителя (строка 11) и ссылки на детей (строка 10). Во втором варианте, когда переданный компонент не является текущим объектом, предполагается, что он – один из его детей. В таком случае программа проходит в цикле по всем дочерним узлам (строки 13–20) и проверяет, не требуется ли удалить какой-либо из них. И если один из детей должен быть удален, то удаляется такой дочерний компонент (строка 17), а также ссылка на него из массива `aChildren` (строка 18).

Метод `remove()` использует функцию `safeRemove()`, показанную в примере 6.6, для полного удаления дочерних компонентов. Эта функция сначала проверяет, является ли переданный ему компонент составным,

и если это оказывается так, то она вызывает его метод `remove()`. Если же переданный ей компонент не составной (а является листовым узлом), то она удаляет его ссылку на родителя.

Пример 6.6. Функция safeRemove()

```
private function safeRemove(c:Component)
{
    if (c.getComposite())
    {
        c.remove(c); // составной
    } else {
        c.removeParentRef();
    }
}
```

Правильная последовательность действий при удалении ссылок на родителя и детей очень важна, чтобы удаленный объект остался полностью изолированным, без входящих или исходящих ссылок. Это является для сборщика мусора необходимым условием для удаления данного объекта и освобождения используемой им памяти.

Построение и манипулирование составной структурой

Давайте протестируем, как работает метод `remove()`, построив составную структуру, показанную в первой колонке на рис. 6.4, и удалив из нее некоторые листья и составные узлы. Со стороны *клиента* выполним все действия, описанные в примере 6.7.

Пример 6.7. Main.as (код клиента по удалению узлов)

```
package
{
    import flash.display.MovieClip;

    public class Main extends MovieClip
    {
        public function Main()
        {
            var root:Composite = new Composite("root");

            //  создаем узел
            var n1:Composite = new Composite("composite 1");
            n1.add(new Leaf("leaf 1")); // добавляем лист
            n1.add(new Leaf("leaf 2")); // добавляем лист
            root.add(n1); //  добавляем узел в корневой

            // создаем другой узел
            var n2:Composite = new Composite("composite 2");
            n2.add(new Leaf("leaf 3")); // добавляем лист
            n2.add(new Leaf("leaf 4")); // добавляем лист
```

```
            n2.add(new Leaf("leaf 5")); // добавляем лист
            root.add(n2); //  добавляем узел в корневой

            // добавляем лист в корневой узел
            root.add(new Leaf("leaf 6"));

            root.operation(); // вызываем operation на корневом узле
            trace("display tree");
            trace("============");
            root.operation();

            trace("remove first child of the second child of root");
            trace("=============================================");
            root.getChild(2).remove(root.getChild(2).getChild(1));
            root.operation();

            trace("remove the second child of root");
            trace("==============================");
            root.remove(root.getChild(2));
            root.operation();        }
    }
}
```

Выполнение действий примера 6.7 должно дать нижеследующий результат. Метод `operation()` **проходит по всей древовидной структуре, используя подход «исследуй первого», и выводит соответствующие имена компонентов.**

```
display tree
============
root
composite 1
leaf 1
leaf 2
composite 2
leaf 3
leaf 4
leaf 5
leaf 6
remove first child of the second child of root
=============================================
root
composite 1
leaf 1
leaf 2
composite 2
leaf 4
leaf 5
leaf 6
remove the second child of root
==============================
root
```

```
composite 1
leaf 1
leaf 2
leaf 6
```

Вы можете легко представить, что происходит с деревом компонентов, посмотрев на рис. 6.5.

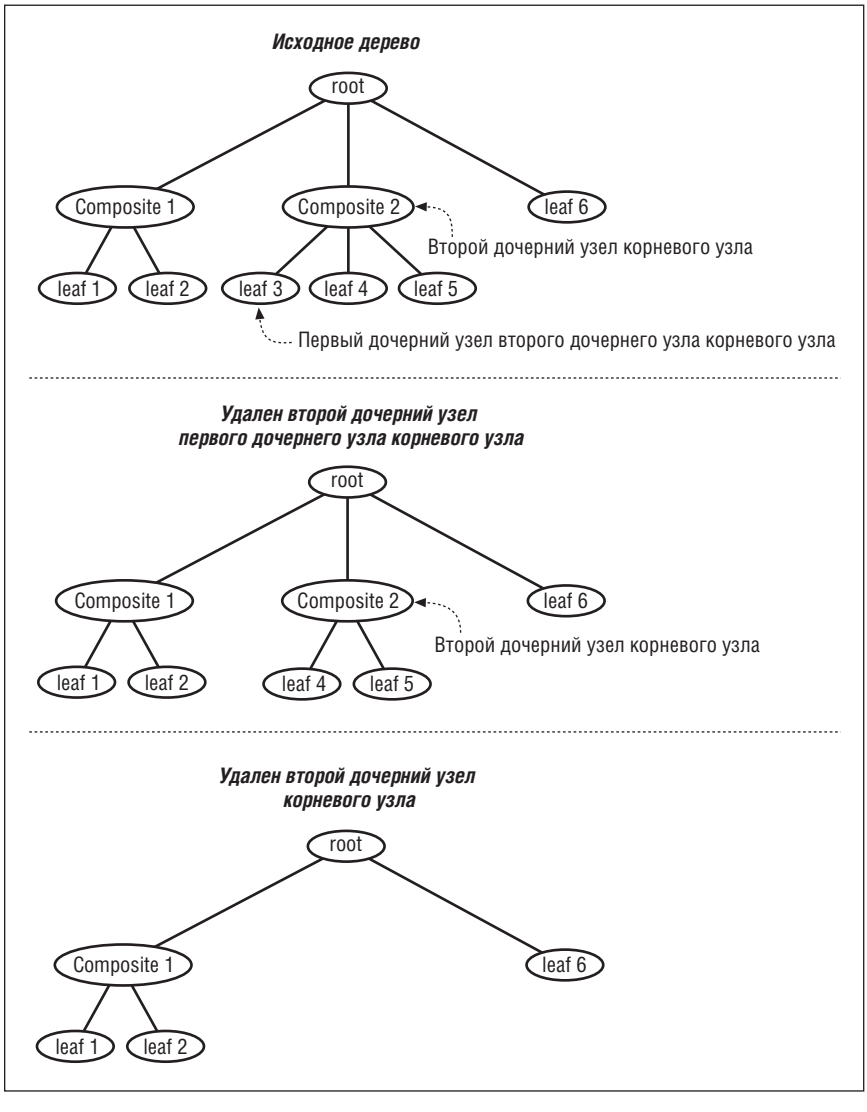


Рис. 6.5. *Удаление компонентов из древовидной структуры*

Ключевые концепции ООП в шаблоне Компоновщик

Ключевой концепцией ООП, неотъемлемой от шаблона Компоновщик, является *полиморфизм*. Полиморфизм может быть в общих чертах описан как способность объектов быть представителями различных классов и по-разному реагировать на вызов одного и того же метода. Это возможно, если сигнатура метода одинакова для данных классов, даже хотя объекты являются представителями различных классов. В двух словах, у методов различных классов должен быть один и тот же интерфейс для клиентов этих классов.

Поскольку шаблон Компоновщик позволяет клиентам рассматривать листовые и составные узлы единым образом, через один интерфейс, он служит хорошим примером использования полиморфизма. Составные и листовые узлы имплементируют единый интерфейс компонента, являясь подклассами его абстрактного класса.

Имплементация операций также является отличным примером использования полиморфизма. В нашем минималистском примере метод `operation()` был имплементирован одним способом в листьях и совершенно другим в составных узлах. Последние были вынуждены передавать вызов данного метода всем своим дочерним узлам. Метод `operation()` по-разному работал в листовых и составных узлах, но клиенту совершенно не требовалось знать об этом. Клиент не видит разных имплементаций, он видит только единый интерфейс.

Пример: списки мелодий

Приходилось ли вам создавать списки любимых мелодий для ваших цифровых проигрывателей? Если да, то делали ли вы следующий шаг и вставляли ли одни такие списки внутрь других для получения еще больших списков? Подобное приложение является отличным примером использования шаблона Компоновщик. Пусть наша музыкальная библиотека содержит в себе мелодии и их списки. Мелодия будет примитивным объектом, а список – составным объектом, содержащим коллекцию мелодий. Создадим сначала класс компонента для нашего примера приложения для списков мелодий.

Файл *Component.as* (пример 6.8) содержит в себе абстрактный класс `Component`, определяющий общий интерфейс для мелодий и их списков. Файл *Song.as* (пример 6.10) содержит класс `Song`, а файл *Playlist.as* (пример 6.9) – класс `Playlist`. Классы `Song` и `Playlist` являются расширением класса `Component` и реализуют в себе необходимые имплементации.

Пример 6.8. Component.as

```
package
{
```

```
    import flash.errors.IllegalOperationError;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class Component
    {
        public function add(c:Component):void
        {
            throw new IllegalOperationError
                ("add operation not supported");
        }

        public function remove(c:Component):void
        {
            throw new IllegalOperationError
                ("remove operation not supported");
        }

        public function getChild(n:int):Component
        {
            throw new IllegalOperationError
                ("getChild operation not supported");
            return null;
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        public function play():void {}
    }
}
```

Класс `Component` **определяет собой абстрактный интерфейс для классов** `Song` **и** `Playlist`**. Он также определяет абстрактный метод** `play()`**.**

Пример 6.9. Playlist.as

```
package
{
    public class Playlist extends Component
    {

        private var sName:String;
        private var aChildren:Array;

        public function Playlist(sName:String)
        {
            this.sName = sName;
            this.aChildren = new Array();
        }

        override public function add(c:Component):void
        {
            aChildren.push(c);
        }
```

```
        override public function play():void
        {
            trace("Queuing playlist: " + this.sName);
            for each (var c:Component in aChildren)
            {
                c.play();
            }
        }
    }
}
```

Класс `Playlist` является расширением класса `Component` и практически идентичен составному классу из нашего минималистского примера.

Очереди мелодий для последовательного проигрывания

Большая часть класса `Song`, показанная в примере 6.10, посвящена работе с последовательностями мелодий и обработке событий. Поскольку мелодии должны проигрываться последовательно, а не одновременно, в данном классе введены несколько статических свойств. В частности, свойство `soundChannel` объявлено *статическим* (`static`), потому что только один канал для проигрывания должен использоваться всеми представителями класса `Song`. Оно гарантирует, что различные объекты мелодий не откроют одновременно многочисленные звуковые каналы для проигрывания музыки. Кроме того, в классе `Song` определяется статическое свойство `aSongQueue`, которое является массивом, содержащим в себе список мелодий для проигрывания. И наконец, статическое свойство `songPlayingFlag` содержит в себе логическую переменную, показывающую, проигрывается ли сейчас какая-либо мелодия.

Пример 6.10. Song.as

```
 1 package
 2 {
 3     import flash.events.*;
 4     import flash.media.Sound;
 5     import flash.media.SoundChannel;
 6     import flash.net.URLRequest;
 7
 8     public class Song extends Component
 9     {
10         private var sName:String;
11         private var song:Sound;
12         private static var soundChannel:SoundChannel =  new SoundChannel();
13         private static var aSongQueue:Array = [];
14         private static var songPlayingFlag:Boolean = false;
15
16         public function Song(sName:String)
17         {
```

```
18              this.sName = sName;
19          }
20
21          override public function play():void
22          {
23              var request:URLRequest = new URLRequest
                 ("music/" + sName);
24              song = new Sound();
25              song.addEventListener(Event.COMPLETE, songLoaded);
26              song.addEventListener(IOErrorEvent.IO_ERROR, loadError);
27              song.load(request);
28          }
29
30          private function songLoaded(event:Event):void
31          {
32              aSongQueue.push(song);
33              playIfIdle();
34          }
35
36          private function loadError(event:Event):void
37          {
38              trace("Error loading song " + this.sName);
39          }
40
41          private function playDone(event:Event):void
42          {
43              songPlayingFlag = false;
44              playIfIdle();
45          }
46
47          private function playIfIdle():void
48          {
49              if (!songPlayingFlag)
50              {
51                  var s:Sound = aSongQueue.shift();
52                  if (s)
53                  {
54                      songPlayingFlag = true;
55                      trace("playing " + s.id3.songName);
56                          // из ID3-тэга
57                      soundChannel = s.play();
58                      soundChannel.addEventListener
                         (Event.SOUND_COMPLETE, this.playDone);
59                  }
60              }
61          }
62      }
63  }
```

Имя файла передается параметризованному конструктору класса и сохраняется в свойстве `sName`. Метод `play()` загружает эти файлы мелодий

из специального подкаталога с именем *media*. Обработчик событий songLoaded() регистрируется на подписку о событии Event. COMPLETE (строка 25). По окончании загрузки мелодии этот обработчик добавляет ее в очередь мелодий на проигрывание aSongQueue (строка 32). После чего он вызывает метод playIfIdle(), который проверяет, проигрывается ли сейчас какая-либо мелодия, и если нет, выбирает мелодию из aSongQueue с помощью функции Shift (строка 51), начинает ее проигрывание и присваивает soundChannel канал проигрывания мелодии (строка 57). В качестве очереди песен на проигрывание используется массив aSongQueue, который работает по принципу «первый пришел – первый ушел». Обработчик событий playDone() регистрируется на подписку о событии Event. SOUND_COMPLETE (строка 58). По окончании воспроизведения мелодии указанное событие запустит обработчик playDone(), который опять вызывет метод playIfIdle().

Теперь мы можем создать несколько списков мелодий на проигрывание и немного послушать музыку.

Построение составных списков мелодий

Создадим два списка мелодий и встроим их в больший список. Следующий код должен выполняться со стороны класса документа Flash.

```
// создаем список проигрывания
var drumlicks:Playlist = new Playlist("drum licks");
drumlicks.add(new Song("bongo.mp3"));
drumlicks.add(new Song("tabla.mp3"));
drumlicks.add(new Song("djembe.mp3"));

// создаем другой список
var guitariffs:Playlist = new Playlist("guitar riffs");
guitariffs.add(new Song("acousticguitar.mp3"));
guitariffs.add(new Song("electricguitar.mp3"));

// создаем составной список
var eclectic:Playlist = new Playlist("eclectic");
eclectic.add(drumlicks);
eclectic.add(new Song("balladpiano.mp3"));
eclectic.add(guitariffs);
eclectic.play();
```

Этот пример приложения вначале создает список мелодий под названием «drum licks» («бой барабанов»), затем список мелодий «guitar riffs» («звуки гитары»). И наконец, оно создает и начинает проигрывать список мелодий под названием «eclectic» («эклектика»), который включает в себя два предыдущих списка и дополнительную мелодию. Следующий выведенный текст показывает последовательность воспроизведения мелодий:

```
Queuing playlist: eclectic
Queuing playlist: drum licks
```

```
Queuing playlist: guitar riffs
playing bongo
playing tabla
playing djembe
playing ballad piano
playing acoustic guitar
playing electric guitar
```

Когда начинает проигрываться список мелодий «eclectic», он добавляет в очередь на воспроизведение и два других списка, «drum licks» и «guitar riffs». Именно такого поведения мы ожидали от структуры составного списка мелодий. Вы можете легко удалить из него мелодии и их списки, реализовав метод remove(), как это было показано в нашем минималистском примере шаблона Компоновщик.

Пример: анимация составного объекта с помощью инверсной кинематики

Существует множество примеров замечательных игр на Flash, использующих ActionScript 3.0 для анимации персонажей на сцене. Даже простая анимационная фигура может иметь независимо функционирующие части тела, такие как руки и ноги, которые могут использоваться для прыжков, бега и ударов. Сможем ли мы разработать сложную анимационную фигуру, рассматривая части ее тела как составные объекты и простые компоненты? Будет ли нам выгодно рассматривать одиночные компоненты и составные части тела нашей фигуры единым образом? На самом деле шаблон Компоновщик предоставляет несколько хороших возможностей для анимации, в чем мы убедимся на примере анимированной змеи, которая будет создана далее. Рисунок 6.6 показывает снимок экрана с такой змеей, движения которой рассчитываются с помощью инверсной кинематики.

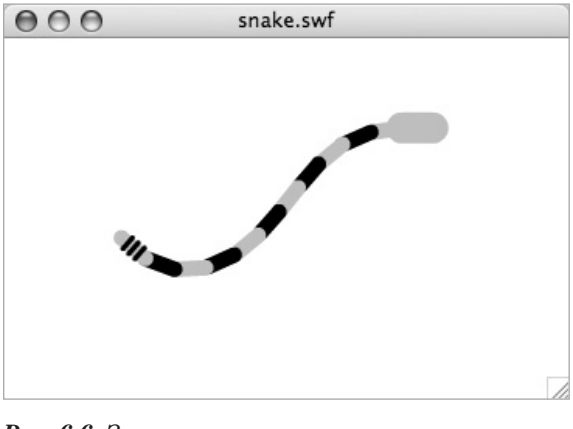


Рис. 6.6. *Змея, сконструированная с помощью шаблона Компоновщик*

Использование инверсной кинематики

Инверсная кинематика – это метод, с помощью которого твердые, связанные между собой объекты могут двигаться, принимая различные позы. Хорошим примером таких объектов является марионетка – кукла, контролируемая кукловодом с помощью веревочек. Руки и ноги марионетки состоят из нескольких частей, соединенных между собой шарнирами. Например, плечо присоединяется к туловищу в районе плечевого сустава. В свою очередь предплечье соединяется с плечом в области локтевого сустава, а кисть руки с предплечьем – в области запястья. Все эти связанные объекты формируют кинематическую цепь. Инверсная кинематика позволяет рассчитывать движения такой цепи, учитывая ограничения, налагаемые всеми суставами. Простейшая форма кинематической цепи имеет свободный конец, положение которого задается извне. Например, у руки марионетки есть свободный конец, так как она прикреплена к веревочке, управляемой кукловодом. Поскольку шарниры соединяют кисть, предплечье и плечо, все они будут двигаться, когда кукловод будет дергать за веревочку. Вы можете попробовать это сами: расслабьте полностью вашу левую руку и потяните ее вверх за кисть вашей правой рукой. Обратите внимание, как внешняя сила двигает вашей левой ладонью, которая передает движение предплечью, которое в свою очередь двигает ваше плечо. Движение подобных связанных объектов, тянущих друг друга, и есть область применения инверсной кинематики. Такой тип движения очень похож на движения змеи в нашей программе, которую мы разработаем.

В нашем учебном приложении пользователь будет контролировать движения головы змеи с помощью клавиатуры. При движении головы все связанные с ней сегменты будут двигаться вслед за ней согласно принципам инверсной кинематики. Таким образом, вся змея будет вести себя как кинематическая цепь.

Кинематические цепи состоят из одной или большего числа кинематических пар. Например, предплечье и плечо – это кинематическая пара. Все связанные шарнирами части тела нашей змеи также образуют кинематические пары. Мы разработаем нашу змею как составной объект. Ее голова будет его корневым узлом, а сегменты ее тела будут подсоединяться один к другому. Каждый сегмент, с которым связан следующий сегмент, будет также составным объектом. Последний сегмент змеи, ее хвост (или погремушка у гремучей змеи), будет простым компонентом.

Создание компонента и составных узлов для змеи

Все наши компоненты будут *отображаемыми объектами* на сцене. Следовательно, нам нужно, чтобы наш класс компонента был подклассом `Sprite`. Это позволит нам унаследовать его возможности по манипуляции нашими компонентами на сцене и обрабатывать нужные нам события. Пример 6.11 содержит наш класс `Component`, который будет нами использоваться для создания анимированных объектов. Его единствен-

ным отличием от аналогичного класса компонента из предыдущего примера является то, что он расширяет класс Sprite и объявляет у себя метод update().

Пример 6.11. Component.as

```
package
{

    import flash.errors.IllegalOperationError;
    import flash.display.Sprite;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class Component extends Sprite
    {

        protected var parentNode:Composite = null;

        public function add(c:Component):void
        {
            throw new IllegalOperationError(
                "add operation not supported");
        }

        public function remove(c:Component):void
        {
            throw new IllegalOperationError(
                "remove operation not supported");
        }

        public function getChild(n:int):Component
        {
            throw new IllegalOperationError(
                "getChild operation not supported");
            return null;
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        public function update():void {}

        internal function setParent(aParent:Composite):void
        {
            parentNode = aParent;
        }

        public function getParent():Composite
        {
            return parentNode;
        }
    }
}
```

Наш составной класс, показанный в примере 6.12, является расширением класса Component и замещает в своей имплементации метод update().

Пример 6.12. Composite.as

```
package
{
    public class Composite extends Component
    {
        protected var aChildren:Array;

        public function Composite()
        {
            this.aChildren = new Array();
        }

        override public function add(c:Component):void
        {
            aChildren.push(c);
            c.setParent(this);
        }

        override public function getChild(n:int):Component
        {
            if ((n > 0) && (n <= aChildren.length))
            {
                return aChildren[n-1];
            } else {
                return null;
            }
        }

        override public function update():void
        {
            for each (var c:Component in aChildren)
            {
                c.update();
            }
        }
    }
}
```

Классы Component и Composite должны рассматриваться как абстрактные классы и использоваться только для создания подклассов. С их помощью мы разработаем классы Head («голова») и BodySegment («сегмент тела») (подклассы класса Composite), а также класс Tail («хвост») (подкласс Component) для создания нашей змеи.

Создание головы змеи

Голова змеи будет корневым классом для нашей составной структуры всей змеи. Каждый компонент змеи будет состоять из линии длиной в 20 пикселов. Пример 6.13 показывает наш класс `Head`, который рисует голову змеи. Его конструктор получает параметр `color` («цвет»), определяющий цвет змеиной головы.

Пример 6.13. Head.as

```
package
{
    public class Head extends Composite
    {
        public function Head(color:uint = 0xC0C0C0)
        {
            graphics.lineStyle(20, color);
            graphics.moveTo(0, 0);
            graphics.lineTo(20, 0);
        }
    }
}
```

Голова змеи будет контролироваться с помощью событий клавиатуры. Следующим шагом мы создадим класс документа, который будет создавать нашу змею и подписываться на события клавиатуры для осуществления контроля ее.

Контроль змеи

В примере 6.14 показан класс `Main`, строящий составную структуру змеи и регистрирующий обработчики событий для контроля ее. Первым делом создается только голова змеи (строки 18–23). Клавиши со стрелками будут заставлять эту голову змеи двигаться вперед, назад, влево и вправо. Голова змеи будет свободным концом кинематической цепи, представляющей собой нашу змею. Подобно веревочке, тянущей кисть марионетки, клавиши клавиатуры будут двигать голову змеи и поворачивать ее.

Пример 6.14. Main.as

```
 1 package
 2 {
 3     import flash.display.Sprite;
 4     import flash.events.*;
 5     import flash.geom.*;
 6     import flash.ui.Keyboard;
 7
 8     /**
 9     *    класс Main
10     *    @ назначение: Класс документа
11     */
```

```
12      public class Main extends Sprite
13      {
14          private var snake:Composite;
15
16          public function Main()
17          {
18              // создаем змею
19              snake = new Head();
20              snake.x = snake.y = 200;
21
22              // ставим змею на сцену
23              addChild(snake);
24
25              // регистрируем на сцене обработчик нажатий клавиш
26              stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPress);
27          }
28
29          private function onKeyPress(event:KeyboardEvent):void
30          {
31              switch (event.keyCode)
32              {
33                  case Keyboard.LEFT :
34                      snake.rotation -= 10;
35                      break;
36                  case Keyboard.RIGHT :
37                      snake.rotation += 10;
38                      break;
39                  case Keyboard.UP :
40                   var newFWLocOffset:Point =
                                Point.polar(5, snake.rotation * Math.PI / 180);
41                      snake.x += newFWLocOffset.x;
42                          //  передвинуться вперед на x-смещение
43                      snake.y += newFWLocOffset.y;
44                          //  передвинуться вперед на y-смещение
45                      break;
46                  case Keyboard.DOWN :
47                      var newBKLocOffset:Point =
                                Point.polar(5, snake.rotation * Math.PI / 180);
48                      snake.x -= newBKLocOffset.x;
49                          //  вернуться назад на x-смещение
50                      snake.y -= newBKLocOffset.y;
51                          //  вернуться назад на y-смещение
52                      break;
53              }
54          }
55      }
56  }[1]
```

[1] **Чтобы новое положение головы змеи «тянуло» за собой сегменты тела, необходимо в конец функции** `onKeyPress` **(после строки 53) добавить вызов функции** `snake.update();`. — ***Прим. науч. ред.***

В строке 26 класса `Main` (пример 6.14) происходит регистрация подписки обработчика `onKeyPress()` на событие клавиатуры `Key_DOWN`. Клавиши со стрелками влево и вправо заставляют голову змеи поворачиваться против и по часовой стрелке на 10 градусов. Клавиши со стрелками вверх и вниз заставляют ее двигаться вперед и назад на 5 пикселов. Нам здесь удобно использовать метод `Point.polar`, поскольку нашей змее нужно двигаться в направлении, куда она смотрит. Этот метод возвращает точку с заданным углом и расстоянием от начала координат (0,0). Добавление ее координат (свойства `x` и `y` класса `Point`) к текущим координатам головы даст нам ее новое положение.

До начала разработки движений сегментов тела змеи нам необходимо изучить, как работает кинематическая цепь.

Движение кинематической пары

Как мы обсуждали ранее, наша змея будет составным объектом, состоящим из нескольких связанных компонентов, которые будут представлять собой кинематическую цепь. Кинематическая цепь составлена из кинематических пар – двух жестких сегментов, соединенных гибким шарниром. Каким образом кинематическая пара может двигаться и поворачиваться? Основная идея заключается в том, что родительский сегмент тянет за собой дочерний сегмент. В простой кинематической паре единственным ограничением на движение является гибкий шарнир. Поскольку он является свободным концом, родительский сегмент будет двигаться и поворачиваться независимо от своего дочернего сегмента. Дочерний сегмент будет стремиться поворачиваться и располагаться так, чтобы быть обращенным к шарниру, а двигаться так, чтобы не отставать от родительского сегмента.

Рисунок 6.7 показывает четыре этапа движения дочернего сегмента вслед за родительским в кинематической паре. Хотя тянет родительский сегмент, всю работу выполняет его потомок.

Ниже приводится описание шагов, изображенных на рис. 6.7.

1. Исходное состояние, при котором кинематическая пара состоит из двух линейных сегментов. Начальной точкой для сегментов является начало их линий (C1 для потомка и P1 для родителя). Исходное расстояние между начальной точкой потомка C1 и шарниром равно d.
2. Родительский сегмент передвигается в позицию P2 и поворачивается (независимо от сегмента потомка).
3. Потомок поворачивается в сторону шарнира.
4. Потомок передвигается к точке P2, чтобы восстановить дистанцию d.

В расчетах подобных движений активно применяется тригонометрия. Однако мы будем просто пользоваться встроенными методами класса

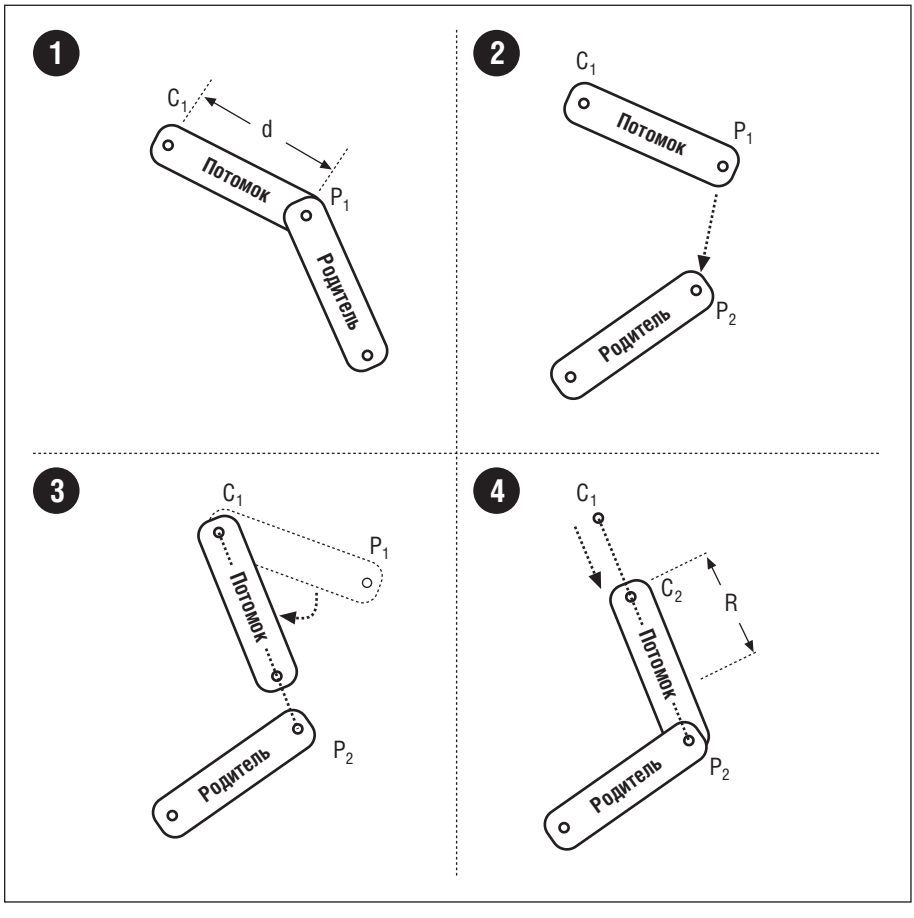


Рис. 6.7. *Движение потомка в связке с родителем в кинематической паре*

`Point` для вычисления всех поворотов и перемещений. Сейчас мы готовы создать сегменты тела нашей змеи.

Построение сегментов тела и хвоста

Нам не нужно волноваться насчет расчетов инверсной кинематики для головы змеи, поскольку она является корневым узлом ее составной структуры. Движения головы змеи задаются с помощью клавиатуры. Однако сегменты тела, которые последовательно прикрепляются к голове, должны двигаться как кинематические пары. Пример 6.15 демонстрирует нам класс `BodySegment`. Он является нашим составным объектом и расширяет класс `Composite`, показанный в примере 6.12. Его

конструктор рисует сегмент тела змеи, а метод `update()` замещен и имплементирует инверсное кинематическое движение.

Пример 6.15. BodySegment.as

```
 1 package
 2 {
 3     import flash.geom.Point;
 4
 5     public class BodySegment extends Composite
 6     {
 7         private var segLen:Number = 20;
 8
 9         public function BodySegment(color:uint = 0xC0C0C0)
10         {
11             graphics.lineStyle(10, color); //  серый цвет
12             graphics.moveTo(0, 0);
13             graphics.lineTo(segLen, 0);
14         }
15
16         override public function update():void
17         {
18             var myParent:Composite = this.getParent();
19             var parentLoc:Point = new Point(myParent.x, myParent.y);
20             var myLoc:Point = new Point(this.x, this.y);
21
22             //  повернуть для ориентации на местонахождение родителя
23             var tempPoint:Point = parentLoc.subtract(myLoc);
24             var angle:Number = Math.atan2(tempPoint.y, tempPoint.x);
25             this.rotation = angle * 180 / Math.PI;
26
27             //  передвинуть для соблюдения дистанции
28             var currentDistance:Number = Point.distance(
29                 parentLoc, myLoc);
30             var myNewLoc:Point = Point.interpolate(myLoc, parentLoc,
                                                segLen / currentDistance);
31             this.x = myNewLoc.x;
32             this.y = myNewLoc.y;
33
34             super.update();
35         }
36     }
37 }
```

Метод `update()` имплементирует здесь шаги 3 и 4, показанные на рис. 6.7. Строки 18–20 получают данные от родителя текущего компонента и создают две точки, которые представляют собой C1 и P2 в шаге 3.

В строках 22–25 происходит вычисление нового угла компонента-потомка, такого, чтобы он был ориентирован на новое положение шар-

нира. Вычитание координат точки C1 из координат P2 в шаге 3 рис. 6.7 дает нам точку, чьи координаты `x` и `y` представляют собой горизонтальное и вертикальное расстояние между двумя точками. Следовательно, новый угол поворота может быть легко вычислен с помощью функции арктангенс от этих двух величин. Обратите внимание, что функция `Math.atan2` возвращает значение угла в *радианах*, и его нужно конвертировать в *градусы* до присвоения свойству `rotation`.

В строках 27–31 производится перемещение корректно ориентированного сегмента-потомка до восстановления нужной дистанции между его центральной точкой и шарниром (она обозначена буквой *d* на рис.6.7). Сначала вычисляется расстояние между точками C1 и P2 (строка 28). Далее функция `Point.interpolation` используется для определения точки C2, в которой будет соблюдаться корректная длина сегмента *d*. Третьим параметром данной функции задается величина отношения (между 0 и 1), определяющая искомую промежуточную точку.

И наконец, метод `update()` вызывает сам себя в суперклассе (строка 33), чтобы провести обновление в его дочерних компонентах.

Одиночный компонент хвост, последний сегмент змеи, показан в примере 6.16. Класс `Tail` является подклассом класса `Component` (пример 6.11). В своем конструкторе он рисует хвост змеи в виде погремушки, превращая нашу обычную змейку в гремучую змею. Кроме того, он имплементирует по-своему метод `update()`, в котором в отличие от класса `BodySegment` не вызывает сам себя в своем суперклассе, поскольку хвост – это одиночный компонент, не имеющий потомков.

Пример 6.16. Tail.as

```
package
{
    import flash.geom.Point;

    public class Tail extends Component
    {

        private var segLen:Number = 20;

        public function Tail(color:uint = 0xC0C0C0)
        {
            graphics.lineStyle(10, color);
            graphics.moveTo(0, 0);
            graphics.lineTo(segLen, 0);
            graphics.lineStyle(3, 0x000000);
            for (var i:uint = 1; i < 4; i++)
            {
                graphics.moveTo(i * 5, -5);
                graphics.lineTo(i * 5, 5);
            }
        }
```

```
        override public function update():void
        {
            var myParent:Composite = this.getParent();
            var parentLoc:Point = new Point(myParent.x, myParent.y);
            var myLoc:Point = new Point(this.x, this.y);

            //  повернуть для ориентации на местонахождение родителя
            var tempPoint:Point = parentLoc.subtract(myLoc);
            var angle:Number = Math.atan2(tempPoint.y, tempPoint.x);
            this.rotation = angle * 180 / Math.PI;

            // передвинуть для соблюдения дистанции
            var currentDistance:Number = Point.distance(
                parentLoc, myLoc);
            var myNewLoc:Point = Point.interpolate(myLoc, parentLoc,
                segLen / currentDistance);
            this.x = myNewLoc.x;
            this.y = myNewLoc.y;
        }
    }
}
```

Вы могли задаться вопросом, почему метод `update()` не был имплементирован в классе `Component`. Мы бы вполне могли там имплементировать самую базовую форму инверсной кинематики и применять ее. Но тогда наша змея стала бы настоящим акробатом, поскольку могла бы свободно вращаться вокруг любого из своих шарниров. Истинная мощь инверсной кинематики реализуется, когда введены предельные углы поворота шарниров. Они накладывают ограничения на поворот сегментов, точно так же, как локоть накладывает ограничения на поворот предплечья. Введение различных предельных углов поворота сделает имплементацию метода `update()` уникальной для каждого сегмента. Следовательно, определение его как *абстрактного* метода в классе `Component` имеет глубокий смысл.

Теперь мы можем нарисовать всю нашу змею.

Построение составной змеи

Пример 6.17 показывает нам измененный конструктор класса `Main` (из примера 6.14), который создает полную змею. Строки 10–25 добавляют к змее сегменты ее тела, чередуя их окраску в серый и черный цвет. В конце добавляется хвост (строки 26–28). Метод `update()` вызывается один раз (в строке 30), чтобы упорядочить и отобразить все компоненты змеи на сцене, поскольку они изначально размещаются вне экрана. Законченная змея должна выглядеть так, как изображенная на рис. 6.6.

Пример 6.17. Измененный конструктор в файле Main.as

```
1 public function Main()
2 {
```

```
    // создаем змею
    snake = new Head();
    snake.x = snake.y = 200;

    // добавляем змею на сцену
    addChild(snake);

    // добавляем сегменты тела
    var parentNode:Composite = snake;
    var color:uint;
    for (var i:uint = 0; i < 10; i++)
    {
        if (i % 2) {
            color = 0x000000; // черный
        } else {
            color = 0xC0C0C0; // серый
        }
        var segment:Composite = new BodySegment(color);
        parentNode.add(segment);
        segment.x = segment.y = -50; //  ставим их за границей экрана
        addChild(segment);
        parentNode = segment;
    }
    var tail:Component = new Tail();
    addChild(tail);
    parentNode.add(tail); // добавляем погремушку

    snake.update(); //  составляем сегменты один за другим

    //  подключаем к сцене обработчик нажатий на клавиши
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPress);
}
```

Обратите внимание, как наша змея была построена с помощью шаблона Компоновщик, путем подсоединения потомков к родительским узлам. Такая конструкция очень проста, поскольку одиночный компонент и составные узлы добавляются в нее одним и тем же способом. Хвост мы добавили вызовом того же метода `add()`, что мы использовали при добавлении составных сегментов тела змеи. Метод `update()` плавно спускается каскадами вниз, проходя по всей структуре, и создает анимационное движение нашей змеи. Еще раз обратим ваше внимание на то, что нам не приходится вызывать разные методы `update()`, чтобы отразить разницу в работе одиночного компонента и составных объектов нашей структуры. Хотя данная простая змея имеет всего одну кинематическую цепь, это приложение может быть расширено на случай многих кинематических цепей с несколькими потомками, прикрепленными к одному родителю. Процесс создания структуры и работа метода `update()` не станут сложнее для такого случая.

Использование встроенной составной структуры Flash: список отображения

В предыдущем примере мы расширяли класс Sprite, чтобы построить составную структуру, отображаемую на сцене. Мы вначале добавляли объект к нашей структуре, а потом добавляли его в *список отображения*, используя метод addChild(). Но что такое список отображения в ActionScript 3.0 приложениях? Он представляет собой древовидную структуру, корневым элементом которой является *сцена* (Stage). Она включает в себя все видимые элементы, которые будут отображаться на этой сцене. Список отображения состоит из двух типов элементов: (1) отображаемых объектов и (2) контейнеров отображаемых объектов. Каждый элемент, отображаемый на сцене, имеет тип *отображаемого объекта*. Напротив, *контейнеры отображаемых объектов* не только имеют свое визуальное представление на сцене, но и могут иметь отображаемые объекты и их контейнеры в качестве своих детей.

Подробное рассмотрение структуры наследования класса Sprite позволяет обнаружить в ней два класса с именами DisplayObject и DisplayObjectContainer.

Класс DisplayObject содержит в себе методы и свойства, которые в основном имеют отношение к визуальному представлению объекта, такие как свойства x и y, задающие его позицию. Класс DisplayObjectContainer, потомок класса DisplayObject, определяет необходимые свойства и имплементирует методы для работы с объектами-потомками (за деталями обращайтесь к документации ActionScript 3.0). Некоторые из методов, имплементированных в классе DisplayObjectContainer, перечислены ниже:

```
addChild(child:DisplayObject):DisplayObject;
getChildAt(index:int):DisplayObject;
removeChild(child:DisplayObject):DisplayObject;
```

Список отображения на самом деле является составной структурой, в которой классы DisplayObject и DisplayObjectContainer представляют одиночные компоненты и составные классы соответственно.

Оба этих класса являются абстрактными и не могут непосредственно создавать своих представителей. Мы должны либо расширять их и определять свои уникальные компоненты, либо применять классы, представляющие собой их потомков. Классы Shape и Bitmap расширяют класс DisplayObject. Более часто используемый класс Sprite является расширением класса DisplayObjectContainer и предпочтительным базовым классом для создания составных объектов.

Тогда возникает очевидный вопрос, сможем ли мы использовать методы по работе с потомками из класса DisplayObjectContainer без их повторной имплементации. На самом деле сможем; и мы создадим составной самолет, чтобы показать, что в некоторых случаях лучше применять встроенные методы работы с потомками.

Создание составного самолета

На рис. 6.8 показан составной самолет, который будет разработан в данном разделе. Мы будем использовать *список отображения* для построения визуальных компонентов. Полезность шаблона Компоновщик будет продемонстрирована нами при автоматическом подсчете веса всего самолета, использующем вес его отдельных компонентов. Кроме того, имплементация шаблона Компоновщик поможет нам следить за повреждениями в каждом компоненте и оценивать общий уровень повреждений самолета. Вы также увидите, как с помощью методов, встроенных в класс `DisplayObjectContainer`, из конструкции могут быть удалены отдельные компоненты, когда их уровень повреждений превысит некоторый критический порог.

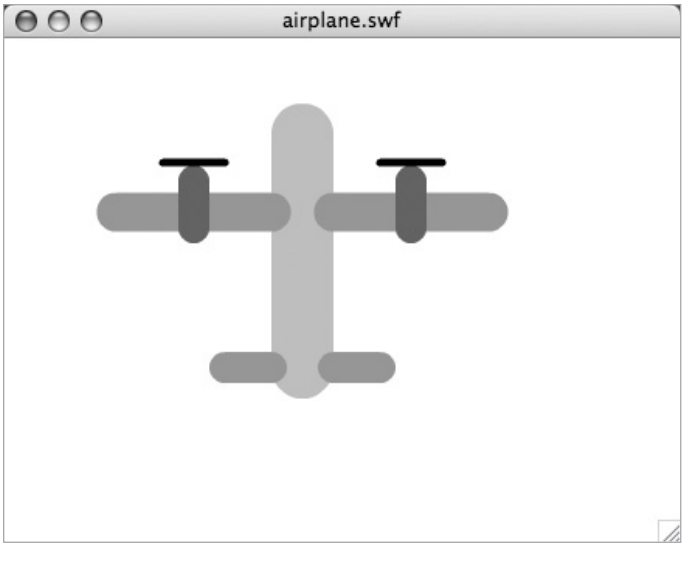


Рис. 6.8. *Двухдвигательный составной самолет (вид сверху)*

Разработка одиночных компонентов и составных классов самолета

Поскольку мы будем использовать встроенные методы класса `DisplayObjectContainer` для манипулирования объектами потомков, имплементация составных классов и одиночных компонентов значительно упростится. Нам потребуется только создать *операции* для нашей составной структуры. Первым шагом надо определить интерфейс операций, возможных для самолета. Пример 6.18 показывает интерфейс `IPlane`. В нем определены две операции `getDamage()` и `getWeight()`, которые возвращают показатели уровня повреждений и вес соответственно.

Пример 6.18. IPlane.as

```
package
{
    public interface IPlane
    {
        function getDamage():Number;
        function getWeight():Number;
    }
}
```

Пример 6.19 содержит класс одиночного компонента самолета Component. Он является расширением класса Shape и имплементирует интерфейс IPlane. Как уже упоминалось ранее, класс Shape является подклассом класса DisplayObject. Конструктор Component принимает в качестве параметров вес и начальный уровень повреждений данного компонента и присваивает их свойствам nWeight и nDamage соответственно. Методы getWeight() и getDamage() просто возвращают значения указанных свойств, поскольку одиночный компонент не имеет потомков.

Пример 6.19. Component.as

```
package
{
    import flash.display.Shape;

    public class Component extends Shape implements IPlane
    {
        protected var nDamage:Number;
        protected var nWeight:Number;

        public function Component(weight:Number, damage:Number = 0)
        {
            this.nDamage = damage;
            this.nWeight = weight;
        }

        public function getDamage():Number
        {
            return nDamage;
        }

        public function getWeight():Number
        {
            return nWeight;
        }
    }
}
```

Здесь важно отметить, что класс Shape не является потомком класса InteractiveObject. Следовательно, класс одиночного компонента самолета

не может реагировать на такие действия пользователя, как щелчки мышью или нажатие клавиш клавиатуры.

Пример 6.20 содержит в себе составной класс нашего самолета Composite. Он является расширением класса Sprite и имплементирует интерфейс IPlane. Класс Sprite расширяет в свою очередь класс DisplayObjectContainer, который представляет собой потомка класса DisplayObject. Таким образом, наша имплементация следует схеме шаблона проектирования Компоновщик. Важным отличием между составным классом и классом одиночного компонента является способ имплементации в них операций. В классе Composite методы getWeight() и getDamage() рекурсивно обходят всех детей и возвращают итоговый вес или уровень повреждений этой составной ветви иерархического дерева. Нам не нужно имплементировать какие-либо методы по работе с детьми, поскольку все они уже унаследованы от класса DisplayObjectContainer.

Пример 6.20. Composite.as

```
package
{
    import flash.display.*;

    public class Composite extends Sprite implements IPlane
    {
        protected var nDamage:Number;
        protected var nWeight:Number;

        public function Composite(weight:Number, damage:Number = 0)
        {
            this.nDamage = damage;
            this.nWeight = weight;
        }

        public function getDamage():Number
        {
            var localDamage:Number = nDamage;
            for (var i:uint = 0; i < this.numChildren; i++)
            {
                var child:DisplayObject = this.getChildAt(i);
                localDamage += IPlane(child).getDamage();
            }
            return localDamage;
        }

        public function getWeight():Number
        {
            var localWeight:Number = nWeight;
            for (var i:uint = 0; i < this.numChildren; i++)
            {
```

```
                var child:DisplayObject = this.getChildAt(i);
                localWeight += IPlane(child).getWeight();
            }
            return localWeight;
        }
    }
}
```

В отличие от класса `Shape` класс `Sprite` является потомком `InteractiveObject`. Следовательно, составные компоненты нашего самолета могут реагировать на события, задаваемые пользователем приложения. Теперь мы можем разработать все одиночные компоненты и составные классы, из которых мы сможем собрать наш самолет.

Создание фюзеляжа, крыльев и двигателей

Теперь, когда у нас есть классы `Component` и `Composite`, мы можем создать все узлы нашего самолета. В примерах с 6.21 по 6.24 показаны классы `Fuselage`, `MainWing`, `TailWing` и `Engine`. Первые два класса, `Fuselage` и `MainWing`, содержат в себе другие компоненты и являются расширениями класса `Composite` (из примера 6.20). Классы `TailWing` и `Engine` являются листовыми узлами и расширяют класс `Component` (из примера 6.19). Имплементации всех этих подклассов очень похожи. Компоненты рисуются простыми линиями с использованием графических свойств класса `DisplayObject`. Первым параметром метода-конструктора является вес компонента, вторым – начальный уровень повреждений (по умолчанию устанавливается в ноль). Обратите внимание на вызов конструктора суперкласса с помощью ключевого слова `super()` в последней строке данного метода-конструктора. Это необходимо для правильной инициализации свойств, определенных в суперклассе.

Пример 6.21. Fuselage.as

```
package
{
    import flash.events.*;

    public class Fuselage extends Composite
    {
        public function Fuselage(weight:Number, damage:Number = 0)
        {
            graphics.lineStyle(40, 0xC0C0C0);
            graphics.moveTo(0, 0);
            graphics.lineTo(0, 150);
            super(weight, damage);
        }
    }
}
```

Пример 6.22. MainWing.as

```
package
{
    public class MainWing extends Composite
    {
        public function MainWing(weight:Number, damage:Number = 0)
        {
            graphics.lineStyle(25, 0x999999);
            graphics.moveTo(0, 0);
            graphics.lineTo(100, 0);
            super(weight, damage);
        }
    }
}
```

Пример 6.23. TailWing.as

```
package
{
    public class TailWing extends Component
    {
        public function TailWing(weight:Number, damage:Number = 0)
        {
            graphics.lineStyle(20, 0x999999);
            graphics.moveTo(0, 0);
            graphics.lineTo(30, 0);
            super(weight, damage);
        }
    }
}
```

Пример 6.24. Engine.as

```
package
{
    public class Engine extends Component
    {
        public function Engine(weight:Number, damage:Number = 0)
        {
            graphics.lineStyle(20, 0x666666);
            graphics.moveTo(0, 0);
            graphics.lineTo(0, 30);
            graphics.lineStyle(5, 0x000000);
            graphics.moveTo(-20, -12);
            graphics.lineTo(20, -12);
            super(weight, damage);
        }
    }
}
```

Хотя наш самолет и нарисован простыми линиями (для понижения степени сложности), мы вполне могли бы создать все компоненты самолета с использованием высокоточных изображений, загружаемых из графических файлов.

Построение составной структуры

На рис. 6.9 показана иерархическая составная структура нашего самолета. Его фюзеляж и несущие крылья являются составными узлами. Фюзеляж содержит в себе два несущих крыла и два хвостовых стабилизатора. А каждое несущее крыло содержит в себе двигатель.

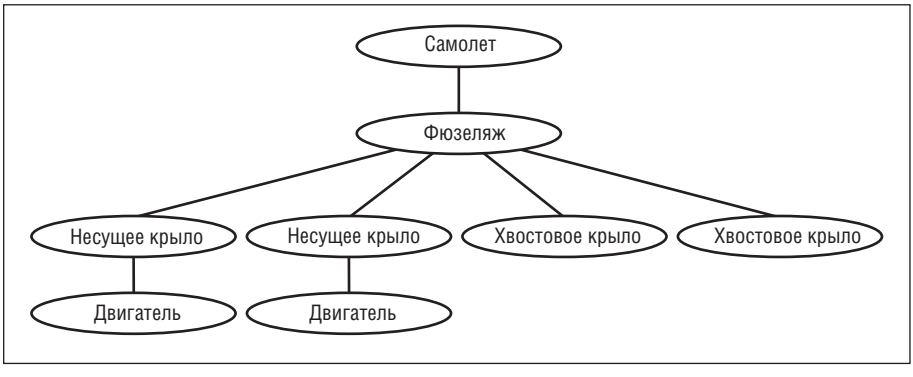


Рис. 6.9. *Иерархическая древовидная структура самолета*

В примере 6.25 приведен класс `Main`, который создает всю структуру нашего самолета. Он должен быть указан в качестве *класса документа* Flash. Самолет изображается, как он виден сверху (см. рис. 6.8). Процедура его построения очень проста и наглядна. Создается представитель каждого компонента самолета и позиционируется относительно начальной точки своего родителя с помощью параметров `x` и `y`. После чего данный компонент добавляется к родительскому композитному узлу посредством метода `addChild()`. Переменная `airPlane` представляет собой корневой узел всего самолета. Вся собранная составная структура самолета добавляется потом к контейнеру отображаемого объекта, связанному с классом `Main` (строка 18). Поскольку класс `Main` является *классом документа* Flash, наш самолет добавляется к *сцене*, которая является корневым узлом списка отображения.

Пример 6.25. Main.as

```
1 package
2 {
3     import flash.display.Sprite;
4
5     /**
6     *    класс Main
```

```
 7     *    @ назначение: Класс документа
 8     */
 9     public class Main extends Sprite
10     {
11         private var airPlane:Composite;
12
13         public function Main()
14         {
15             // создаем самолет
16             airPlane = new Composite(0.0);
17             airPlane.x = 250; airPlane.y = 100;
18             addChild(airPlane);
19
20             // добавляем фюзеляж
21             var fuselage:Composite = new Fuselage(1000, 0)
22             airPlane.addChild(fuselage);
23
24             //  добавляем главное крыло по левому борту
25             var rightWing:Composite = new MainWing(200, 0);
26             rightWing.x = 20; rightWing.y = 50;
27             fuselage.addChild(rightWing);
28
29             //  добавляем главное крыло по правому борту
30             var leftWing:Composite = new MainWing(200, 0);
31             leftWing.scaleX = -1; // отобразить зеркально
                                     относительно вертикальной оси
32             leftWing.x = -20; leftWing.y = 50;
33             fuselage.addChild(leftWing);
34
35             // добавляем двигатель на правое крыло
36             var rightEngine:Component = new Engine(300, 0);
37             rightEngine.x = 50; rightEngine.y = -20;
38             rightWing.addChild(rightEngine);
39
40             // добавляем двигатель на левое крыло
41             var lefttEngine:Component = new Engine(300, 0);
42             lefttEngine.x = 50; lefttEngine.y = -20;
43             leftWing.addChild(lefttEngine);
44
45             // добавляем хвостовое крыло справа
46             var leftTailWing:Component = new TailWing(50, 0);
47             leftTailWing.scaleX = -1; // отобразить зеркально
                                         относительно вертикальной оси
48             leftTailWing.x = -20; leftTailWing.y = 150;
49             fuselage.addChild(leftTailWing);
50
51             // добавляем хвостовое крыло слева
52             var rightTailWing:Component = new TailWing(50, 0);
53             rightTailWing.x = 20; rightTailWing.y = 150;
54             fuselage.addChild(rightTailWing);
55
```

```
56              trace('Weight of airplane: ' + airPlane.getWeight());
57                  // общий вес
58          }
59      }
60  }
```

Подсчет общего веса самолета

Строка 56 класса `Main` из примера 6.25 вызывает метод `getWeight()` на корневом узле нашего самолета. Результат работы этого выражения будет следующим:

```
Weight of airplane: 2100
```

Метод `getWeight()` вычисляет вес самолета рекурсивно, добавляя к нему вес всех компонентов из структуры самолета. Это очень мощный метод подсчета общего параметра в сложной составной структуре. Если бы наш самолет использовался в игре, его вес был бы очень важен для создания реалистичного движения самолета, учитывающего законы физики. Вес самолета может меняться динамически. Например, наш самолет может везти в себе пассажиров, бомбы и дополнительные топливные баки. Посадка на него пассажиров добавит ему веса, сброс бомб резко его уменьшит, а вес топлива будет постепенно снижаться во время всего полета. Метод `getWeight()` будет работать точно таким же образом, если мы будем добавлять и удалять из самолета другие компоненты, такие как пассажиры, бомбы и т. д.

Учет уровня повреждений в компонентах

Теперь мы модифицируем все составные компоненты нашего самолета, добавив в них свойство, которое будет учитывать их уровень повреждения и будет увеличиваться в случае щелчка по какому-либо из них мышкой. В более реалистичном сценарии игры источником повреждений будут снаряды, ракеты и случайные поломки в этих компонентах. Однако для простоты мы представим, что щелчок мышью – это что-то вроде попадания снаряда. В примере 6.26 мы модифицируем класс `Fuselage` (из примера 6.21), чтобы он реагировал на щелчки мышью.

Пример 6.26. Файл Fuselage.as с модифицированным классом Fuselage

```
 1 package
 2 {
 3     import flash.events.*;
 4
 5     public class Fuselage extends Composite
 6     {
 7         public function Fuselage(weight:Number, damage:Number = 0)
 8         {
 9             graphics.lineStyle(40, 0xC0C0C0);
10             graphics.moveTo(0, 0);
11             graphics.lineTo(0, 150);
```

```
12
13              addEventListener(MouseEvent.CLICK, doDamage);
14
15              super(weight, damage);
16          }
17
18          private function doDamage(evt:Event)
19          {
20              this.nDamage += 10;
21              trace('Damage to the fuselage is now ' + this.nDamage);
22              evt.stopPropagation(); // прервать дальнейшее распространение
события
23                  // к последующим узлам
24          }
25      }
26  }
```

Строка 13 в модифицированном конструкторе класса `Fuselage` (пример 6.26) регистрирует метод `doDamage()` как обработчика событий щелчков мышью. Этот метод просто увеличивает свойство `nDamage`, определенное в подклассе, на 10. Вызов события `stopPropagation()` в строке 22 очень важен, поскольку позволяет изолировать ответы на событие мыши только данным отображаемым объектом. Если такой вызов не будет произведен, то это событие будет распространяться далее по всем родительским узлам данного объекта в списке отображения и обрабатываться их зарегистрированными обработчиками событий. Вызов события `stopPropagation` гарантирует, что только самый нижний узел в иерархии списка отображения, у которого есть зарегистрированный обработчик этого события, совершит его обработку. В документации по ActionScript 3.0 рассказывается о распространении событий более подробно. В примере 6.27 мы модифицируем класс `MainWing` (из примера 6.22), чтобы воспроизвести вышеописанное поведение.

Пример 6.27. Файл MainWing.as с модифицированным классом MainWing

```
1  package
2  {
3      import flash.events.*;
4
5      public class MainWing extends Composite
6      {
7          public function MainWing(weight:Number, damage:Number = 0)
8          {
9              graphics.lineStyle(25, 0x999999);
10             graphics.moveTo(0, 0);
11             graphics.lineTo(100, 0);
12
13             addEventListener(MouseEvent.CLICK, doDamage);
14
15             super(weight, damage);
```

```
16          }
17
18          private function doDamage(evt:Event)
19          {
20              this.nDamage += 20;
21              trace('Damage to this wing is now ' + this.nDamage);
22              if (this.nDamage > 50)
23              {
24                  trace('Wing detached from fuselage - fatal crash!');
25                  parent.removeChild(this);
26              }
27              evt.stopPropagation();
28                  // прекратить дальнейшее распространение события
29          }
30      }
31  }
```

В данном примере модифицированного класса `MainWing` обратите особое внимание на условное выражение в строках 22–26. Если уровень повреждений для узла превышает число 50, то этот узел удаляет сам себя из списка отображения (и из составной структуры самолета) с помощью вызова `removeChild()` в родительском узле. До того как мы сможем протестировать наше модифицированное приложение, нам необходимо сделать несколько изменений в классе `Main`, чтобы он отображал информацию об уровне повреждений.

Вычисление общего уровня повреждений самолета

Чтобы упростить работу нашего примера, мы зарегистрируем обработчик щелчка мыши с именем `showDamage`, который будет выводить общий уровень повреждений самолета, для всей *сцены*. Для этого нужно добавить следующее выражение в конец конструктора класса `Main`, показанного в примере 6.25.

```
stage.addEventListener(MouseEvent.CLICK, showDamage);
```

Кроме того, надо добавить и сам обработчик события `showDamage` в класс `Main`.

```
private function showDamage(evt:Event)
{
    trace('Total damage: ' + airPlane.getDamage());
}
```

Теперь щелчок мышью по сцене будет выводить на панель `Output` общий уровень повреждений самолета. А щелчки мышью по фюзеляжу или по несущим крыльям самолета будут выводить текущий уровень повреждений этих компонентов. Ниже показан пример того, что может быть получено в результате нескольких щелчков мышью:

```
Damage to this wing is now 20              // щелчок по правому крылу
Damage to this wing is now 40              // щелчок по правому крылу
```

```
Damage to the fuselage is now 10            // щелчок по фюзеляжу
Damage to the fuselage is now 20            // щелчок по фюзеляжу
Damage to this wing is now 20               // щелчок по левому крылу
Total damage: 80                            // щелчок по сцене
Damage to this wing is now 60               // щелчок по правому крылу
Wing detached from fuselage - fatal crash!
```

Правое крыло было удалено из списка отображения, как только его уровень повреждений превысил 50. В идеале это крыло должно было не исчезнуть, а отвалиться. Если бы у каждого компонента были подпрограммы для автономного движения, то можно было бы компонент крыла удалить от фюзеляжа и сделать дочерним узлом сцены, где оно бы падало под действием гравитации и других имитированных физических эффектов.

Использование списка отображения в ActionScript 3.0 для создания составных структур имеет множество преимуществ благодаря своей простой интеграции с моделью объектов документа Flash. Этот метод предпочтителен, когда составной объект жесткий (все его части движутся как единое целое). Самолет является жестким объектом, даже хотя он и состоит из множества компонентов. Изменение свойств `x` и `y` у корневого узла самолета будет двигать его целиком. Однако этот метод не очень удобен, когда компоненты двигаются независимо друг от друга, как в примере со змеей. В том случае положение каждого из компонентов зависело не от положения корневого узла всей структуры, а от положения родительского узла. Если бы такая составная структура была реализована с помощью списка отображения, для ее работы потребовались бы дополнительные геометрические трансформации.

Заключение

Шаблон проектирования Компоновщик позволяет вам строить сложные системы, состоящие из нескольких меньших компонентов. Компоненты, составляющие систему, могут быть как отдельными компонентами, так и контейнерами, содержащими наборы других компонентов. Основным преимуществом шаблона Компоновщик является то, что он дает клиентам возможность работать с отдельными компонентами (листовыми узлами) и с составными компонентами (составными узлами) одним и тем же способом, через единый интерфейс. Этот шаблон особенно полезен при использовании в ActionScript 3.0, поскольку его список отображения уже имплементирует шаблон Компоновщик, позволяющий разработчикам легко строить и манипулировать сложными отображаемыми объектами.

IV

Поведенческие шаблоны

Поведение людей станет понятным, если вы рассмотрите его с точки зрения их целей, потребностей и мотивов.

Томас Манн

Поведение, которое внешне кажется корректным, но по своей сути является порочным, всегда возмущает тех, кто смотрит вглубь вещей.

Джеймс Брайант Конант (Замечание по поводу забывчивости имплементировать один из методов интерфейса.)

Как следует из нашего опыта, международный закон обеспечивает нам стабильность и порядок, в том смысле, что он позволяет предсказывать поведение тех, с кем мы имеем взаимные законные обязательства.

Джеймс Уильям Фулбрайт (Объяснение, почему вам следует избегать глобальных переменных при реализации шаблонов проектирования.)

Не было никакой разницы между действиями Бога и работой простого случая.

Томас Пинчон (Описание попытки отладить программу.)

Поведенческие шаблоны работают в области взаимодействия между классами и объектами, а также при распределении ответственности. В них описываются как шаблоны для классов и объектов, так и взаимодействие между ними. В шаблонах поведения вы обнаружите гораздо больше разделения труда между участниками, чем во всех остальных шаблонах. Вместо того чтобы определенный класс выполнял какую-то

задачу, эта задача, как правило, инкапсулируется и потом используется классом или многими классами. Вы увидите, что в шаблонах поведения имеется тенденция распределять алгоритмы работы и зоны ответственности между различными объектами. Рисунок IV.1 иллюстрирует это общее описание.

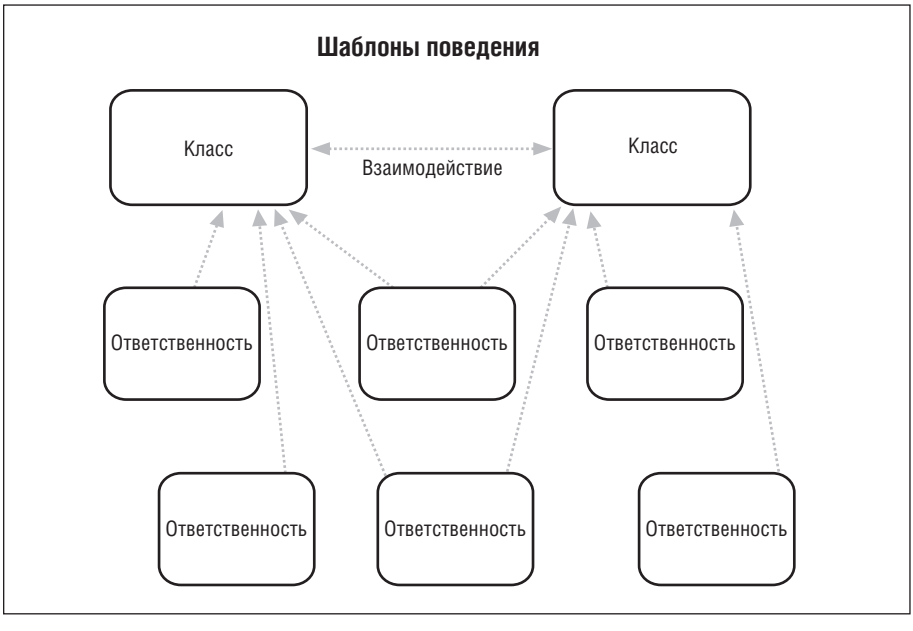


Рис. IV.1. *Шаблоны поведения распределяют ответственность*

Главной задачей шаблонов поведения является распределение ответственности через наследование или композицию. На самом деле только *Шаблонный метод* из данной части книги использует наследование для распределения ответственности. Это простой шаблон проектирования, который использует абстрактный класс для общего описания алгоритма, а его подклассы – для детальной реализации всех аспектов действий. Гораздо более типичным для шаблонов поведения является применение композиции для распределения ответственности. В шаблонах *Команда*, *Наблюдатель*, *Состояние* и *Стратегия* для распределения задач между классами и объектами используется композиция.

Глава 7 «Шаблон Команда»

Глава 8 «Шаблон Наблюдатель»

Глава 9 «Шаблонный метод»

Глава 10 «Шаблон Состояние»

Глава 11 «Шаблон Стратегия»

7

Шаблон Команда

Командование кем-то – это служение,
не больше и не меньше.

Андре Мальро

Если вы будете делать обычные вещи необычным
способом, вы привлечете к себе внимание всего мира.

Джордж Вашингтон Карвер

Создавай как бог, командуй как король, работай как раб.

Константин Бранкузи

Что такое шаблон Команда?

Шаблон Команда позволяет клиенту выдавать запрос на исполнение некоторого поручения объекту, не имея никакого представления об этом объекте и способе выполнения им данного поручения. Можно представлять себе такой запрос как некую посылаемую объекту команду выполнить какое-то известное действие. Самым простым способом осуществления запроса является создание представителя объекта и вызов у него метода, который как-то имплементирует требуемую команду (или поведение). Например, предположим, что мы строим дом, в котором компьютер может управлять многими его компонентами, такими как освещение, двери, обогреватели и т. д. Давайте посмотрим на код, включающий нам свет. Пусть в классе `Light` («свет») имплементирован метод `on()` («включить»), который зажигает свет в доме. Тогда клиенту нужно выполнить следующий код, чтобы включить свет:

```
var light = new Light();
light.on();
```

Давайте посмотрим на другую команду – открыть дверь. *Получателем* этой команды является уже представитель класса `Door` («дверь»), в котором имплементирован метод `open()` для открывания двери.

```
var frontdoor = new Door();
frontdoor.open();
```

Обратите внимание на тесную взаимосвязь между клиентом и получателем команды. Под взаимосвязью мы понимаем степень зависимости одного фрагмента кода от другого. Здесь клиент жестко привязан не только к классам получателя (`Light` и `Door`), но и к их конкретным методам (`on()` и `open()`) в этих классах. Такая ситуация не очень хорошая, если мы хотим построить гибкую систему, способную к расширению в будущем.

Что произойдет, например, если мы заменим нашу обычную дверь раздвижной дверью? А если новым классом, контролирующим эту дверь, будет `SlidingDoor` («раздвижная дверь»), а его новым методом для ее открытия – `slideOpen()` («раздвинуть дверь»)? Нам придется тогда модифицировать код клиента, чтобы указать в нем новые классы получателя команд. А мы договорились всячески избегать ситуаций, в которых требуется модификация существующего кода. Вдобавок эта новая ситуация может потребовать модификаций во многих местах. Например, если входная дверь контролируется с двух точек, с помощью настенной панели, где каждому объекту предназначается своя кнопка, и с помощью переносного пульта (типа пульта телевизора), то нам придется менять классы получателя команд для входной двери в обоих этих устройствах контроля. Кроме того, вы не сможете поменять кнопки управления для другого устройства, поскольку внутренние коды управления жестко прописаны для каждой кнопки.

Чтобы наша система была гибкой и расширяемой, команды должны приписываться кнопкам управления без конкретного указания получателя команды и его конкретного метода ее выполнения. Это позволило бы нам удалить жесткую связь между клиентом и получателем команды. Однако как же такое осуществить? Возможно, наш ответ покажется вам сначала не очень понятным, но нам нужно инкапсулировать получателя команды и его метод в отдельный командный объект. Под инкапсуляцией мы понимаем сокрытие конкретного получателя команды и его метода от того, кто его вызывает. Давайте рассмотрим нетехнический пример, чтобы понять, как этот командный объект может выглядеть.

Несколько поручений от мамы

Родители часто дают какие-то поручения детям по дому, чтобы занять их. Таким образом дети приучаются выполнять свою часть работы по

дому. Попросить детей что-то сделать очень просто – надо их только попросить. Однако будут ли они делать то, что их попросили, уже тема совсем другого разговора. В нашем примере мы будем иметь дело с образцовыми детьми, которые очень послушны и выполняют все порученные им задания без каких-либо возражений. Предположим, что мама придумала для каждого человека в доме небольшое поручение. Однако ей нужно на день уехать в командировку, и она не сможет дать эти поручения им лично. Маме необходимо как-то формализовать процедуру назначения заданий для всех случаев, когда ее не будет на месте. Это отличная возможность для имплементации структуры шаблона проектирования Команда.

Итак, у мамы есть несколько поручений по дому. Она решила написать на бумажках поручения и то, кому они даются. Папа просмотрит эти записки и передаст их конкретным детям на исполнение. Поскольку папа известен своей способностью терять записки и напоминания, мама сделала назначение заданий более официальным и вложила все записки с их описаниями в конверты. Такая записка является аналогом *интерфейса команды*, т. е. просто интерфейса, определяющего метод (часто называемый `execute` («выполнить»)), с помощью которого выполняется определенная задача. Рисунок 7.1 показывает такой домашний эквивалент интерфейса команды. Интерфейс команды от мамы состоит из записки с ключевым словом «сделать», в которой описано, что нужно сделать по дому и кому именно.

Рис. 7.1. *Интерфейс команды мамы*

Мама создает несколько *конкретных команд*, удовлетворяющих *интерфейсу команды*, для тех дел, которые нужно сделать по дому, пока ее не будет. Она кладет записки с конкретными поручениями для каждого человека в четыре разных конверта. Когда эти конверты запечатаны, невозможно сказать, какому ребенку дается поручение и какое именно. Все, что мы знаем, – это то, что в конверте есть поручение. Таким образом, получатель команды и само задание скрыты, или *инкапсулированы*, в конверте.

Рисунок 7.2 показывает конкретные команды от мамы, которые имплементируют ее интерфейс команды, описанный на рис. 7.1. Вот четыре конкретные команды: Джон должен *помыть посуду*, Джейн – *выгулять собаку*, Джек – *запустить стирку*, а папа – *убрать в гараже* (он пока не знает, что ему предстоит). Мама назначает каждое задание

тому, кто больше всех подходит для его выполнения. Она знает, что выгул собаки стоит поручить Джейн, поскольку Брут лучше всего ведет себя, когда именно Джейн рядом с ним. Папа – лучший кандидат на уборку в гараже, ведь, в конце концов, это он развел там беспорядок, и так далее, и так далее.

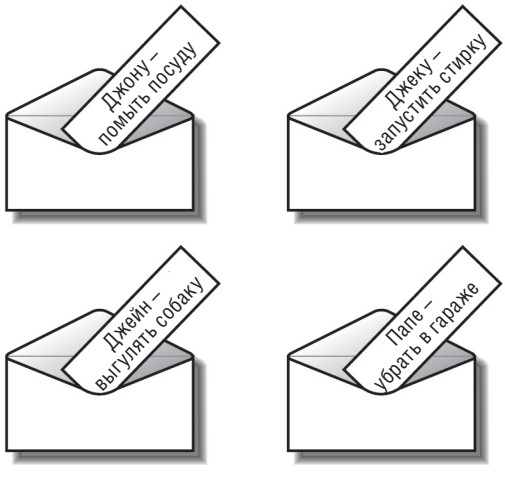


Рис. 7.2. *Домашний эквивалент конкретных команд*

Каждый из таких конвертов инкапсулирует в себе конкретное поручение конкретному исполнителю. Конверты портативны, и их можно просто передать кому-либо (папе), кто попросит назначенных людей выполнить указанные в них поручения. Мама отдает запечатанные конверты папе, который будет исполнять роль *диспетчера*. Он будет хранить у себя все конверты, пока не придет время исполнять поручения. Папа не знает, какие поручения содержатся в этих конвертах, кто их будет исполнять и каким образом. Он только знает, что надо открыть запечатанные конверты и прочитать инструкции, что нужно *сделать* («Джону, *помыть посуду*»; «Джеку, *запустить стирку*» *и т. д.*). Таким образом, мы разделили *получателя* команды и *метод*, которым он будет ее исполнять, путем их инкапсуляции в *объект команды*, которым является запечатанный конверт.

Когда папа привозит детей из школы, наступает время выполнения поручений. Он открывает каждый конверт, называет содержащиеся в них задания для каждого ребенка, а потом отправляется выполнять то, что поручено ему (что-то ворча себе под нос). Папа не имеет понятия, как конкретно дети будут выполнять порученные им дела. Джейн поедет на велосипеде, когда будет выгуливать пса. Джон попросит своего друга Майка помочь ему загрузить посуду в посудомоечную машину. То, как каждый *получатель* команды будет ее исполнять, *диспетчера* совершенно не касается.

Ключевые особенности шаблона Команда

Главным полезным свойством шаблона Команда является то, что он позволяет задавать нужное поведение в программе, сохраняя при этом ее гибкость и расширяемость.

- Шаблон Команда инкапсулирует нужное поведение в портативный командный объект.
- Шаблон Команда отделяет конкретные классы и их методы, осуществляющие нужные действия, от того места, где эти действия вызываются.
- Шаблон Команда позволяет клиенту динамически создавать новое поведение путем создания новых командных объектов и передачи их диспетчеру во время выполнения программы.
- Шаблон Команда позволяет упростить имплементацию командных цепочек, отмены команды, повторения команды и ведения журнала команд в приложении.

Диаграмма классов шаблона Команда

Класс команды `Command` (рис. 7.3) – это интерфейс, в котором определен как минимум один метод `execute()` («выполнить»). Классы `ConcreteCommand` имплементируют интерфейс `Command`, и их может быть несколько. В конкретных командах обычно бывает *параметризованный конструктор*, получающий представителя класса получателя команды для имплементации нужного поведения. Клиент создает представителя такого класса `Receiver` («получатель») и передает его конструктору `ConcreteCommand` при создании новой конкретной команды.

Класс `ConcreteCommand` хранит в себе ссылку на объект получателя и делегирует ему работу при имплементации метода `execute()`.

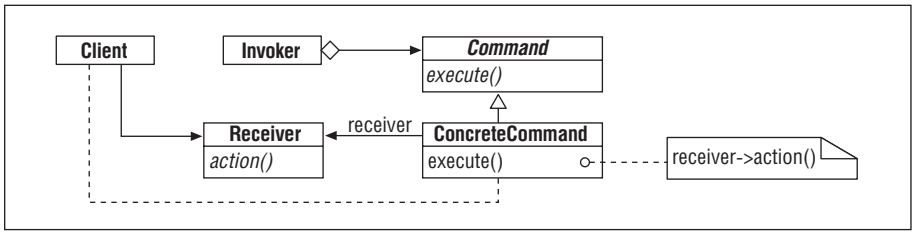


***Рис. 7.3.** Диаграмма классов шаблона Команда*

Клиент передает каждого представителя `ConcreteCommand` специфическим триггерам в объектах классов *диспетчеров*. Диспетчеры занимается тем, что вызывают нужные команды. Они хранят в себе объекты `ConcreteCommand` и вызывают их метод `execute()`, когда наступает время выполнения данной команды. Вам будет понятно, как это все имплементируется на ActionScript 3.0, из следующего минималистского примера.

Минималистский пример шаблона Команда

В этом примере мы имплементируем диаграмму классов шаблона Команда из рисунка 7.3. Шаблон Команда состоит из *интерфейса команды*; *конкретных команд*, которые имплементируют этот интерфейс; *диспетчеров*, которые вызывают метод `execute()` конкретных команд; *получателей*, которые имплементируют указанное в команде поведение, и *клиентов*, которые создают конкретные команды и передают их диспетчерам.

Примеры с 7.1 по 7.5 содержат в себе код минималистской имплементации шаблона проектирования Команда.

Интерфейс команды

Пример 7.1 содержит класс `ICommand`, который определяет интерфейс для команд. В нем определяется единственный метод с именем `execute()`.

Пример 7.1. ICommand.as

```
package
{
    public interface ICommand {
        function execute():void;
    }
}
```

Конкретная команда

Пример 7.2 содержит класс `ConcreteCommand`, который имплементирует интерфейс `ICommand`. Параметризованный конструктор этого класса получает представителя класса `Receiver` и присваивает его своему свойству `receiver`. Выполнение команды методом `execute()` имплементировано передачей полномочий по ее исполнению объекту *получателя* путем вызова его метода `action()`. Обратите внимание, что поскольку представитель получателя передается конструктору, классы `ConcreteCommand` и `Receiver` слабо связаны, и это позволяет при необходимости передавать подкласс класса `Receiver`.

Пример 7.2. ConcreteCommand.as

```
package
{
    class ConcreteCommand implements ICommand
    {
        var receiver:Receiver;

        public function ConcreteCommand(rec:Receiver):void
        {
            this.receiver = rec;
        }
```

```
        public function execute():void
        {
            receiver.action();
        }
    }
}
```

Получатель

Пример 7.3 содержит класс `Receiver`. Он имплементирует метод с именем `action()`. В шаблоне проектирования Команда классы получателя реализуют в себе требуемое командой поведение. Элементами, которым известно о получателях в данном шаблоне, являются конкретные команды и клиент. Получатели скрыты от диспетчеров.

Пример 7.3. Receiver.as

```
package
{
    class Receiver
    {
        public function action()
        {
            trace("Receiver: doing action");
        }
    }
}
```

Диспетчер

Пример 7.4 содержит класс `Invoker`. В нем имеется метод `setCommand()`, получающий в качестве параметра представителя класса конкретной команды, который сохраняется в свойстве `currentCommand`. Метод `executeCommand()` вызывает метод `execute()` представителя конкретной команды. Обратите внимание, что в диспетчере нет ссылки на получателя, и он не имеет понятия о его типе.

Пример 7.4. Invoker.as

```
package
{
    class Invoker
    {
        var currentCommand:ICommand;

        public function setCommand(c:ICommand):void
        {
            this.currentCommand = c;
        }

        public function executeCommand()
```

```
        {
            currentCommand.execute();
        }
    }
}
```

Клиент

Пример 7.5 содержит класс `Main`, представляющий собой *клиента* (и являющийся также *классом документа* Flash). Клиент выполняет несколько действий. Сначала он создает представителя класса *получателя* (строка 9) и передает его как параметр при создании представителя `ConcreteCommand` (строка 10). Представитель класса `ConcreteCommand` называется *командным объектом*. Затем клиент создает представителя класса диспетчера `Invoker` (строка 12) и передает ему командный объект (строка 13). Наконец, клиент выполняет команду, вызывая метод `execute()` командного объекта.

Пример 7.5. Main.as

```
 1 package
 2 {
 3     import flash.display.MovieClip;
 4
 5     public class Main extends MovieClip
 6     {
 7         public function Main()
 8         {
 9             var rec:Receiver = new Receiver();
10             var concCommand:ICommand = new ConcreteCommand(rec);
11
12             var invoker:Invoker = new Invoker();
13             invoker.setCommand(concCommand);
14             concCommand.execute();[1] // Выполнить команду
15         }
16     }
17 }
```

Результатом работы такого минималистского приложения будет следующая распечатка от объекта получателя, показывающая, что был вызван его метод `action()`.

```
Receiver: doing action
```

Установка триггера вызова команды

В большинстве ситуаций клиент не вызывает метод `execute()` в командном объекте. В таком случае диспетчер был бы ему просто не нужен.

---

[1] По логике здесь следовало бы стоять команде `invoker.executeCommand()`. — *Прим. науч. ред.*

Диспетчеры хранят у себя командные объекты, пока не придет время их исполнять. Для запуска действий может быть много триггеров, таких как таймеры и события от пользователя приложения.

Чтобы в нашем минималистском примере проявилась истинная сущность диспетчера, мы можем имплементировать в нем временное событие, которое вызывет исполнение команды. Пример 7.6 содержит в себе такой класс `TimedInvoker`, который расширяет класс `Invoker` (см. пример 7.4). В нем имплементируется метод `setTimer()`, который создает таймер, выдающий временные события 5 раз с интервалом в одну секунду между событиями (1000 тактов («ticks») равно 1 секунде) (строка 10). Потом он регистрирует обработчик для данного временного события `onTimeEvent()` (строка 11) и запускает созданный таймер. Метод `onTimeEvent()` вызывает метод `executeCommand()` в своем суперклассе.

Пример 7.6. TimedInvoker.as

```
 1 package {
 2
 3     import flash.events.Event;
 4     import flash.events.TimerEvent;
 5     import flash.utils.Timer;
 6
 7     class TimedInvoker extends Invoker {
 8
 9         public function setTimer() {
10             var timer:Timer = new Timer(1000, 5);
11             timer.addEventListener(TimerEvent.TIMER, this.onTimerEvent);
12             timer.start();
13         }
14
15         public function onTimerEvent(evt:TimerEvent):void {
16             this.executeCommand();
17         }
18     }
19 }
```

Замените строчки с 12 по 14 в классе `Main` (из примера 7.5) следующими выражениями, чтобы задействовать новый временной диспетчер.

```
var invoker:TimedInvoker = new TimedInvoker();
invoker.setCommand(concCommand);
invoker.setTimer();
```

Это заставит команду исполняться 5 раз с интервалом в секунду, синхронно с временными событиями. Такое приложение точнее представляет шаблон проектирования Команда, в котором диспетчер исполняет команды, основываясь на различных триггерах ситуаций, независимо от клиента.

Ключевые концепции ООП в шаблоне Команда

Ключевой концепцией шаблона Команда является инкапсуляция. Инкапсуляция – это, по существу, сокрытие информации. Вы хотите скрыть детали имплементации некоторых частей программы, которые могут быть изменены в других ее частях.

Командный объект, являющийся представителем класса конкретной команды, содержит в себе поручение. Однако то, какие классы исполняют данное поручение и какие методы в этих классах его имплементируют, полностью скрыто от того, кто вызывает исполнение данного поручения. Эта информация инкапсулирована внутри командного объекта.

Как мы видели в нашем минималистском примере, нигде в классе диспетчера (из примера 7.4) не упоминался тип исполнителя поручения. Диспетчер только знает, что реализовано в интерфейсе команды (пример 7.1). Он лишь знает, что у командного объекта есть метод `execute()` и его нужно вызвать, когда появятся условия (или наступит время), чтобы сделать это.

Такое положение дел полностью развязывает между собой диспетчера и получателя команды. Если нам становится нужным поменять конечного исполнителя для реализации требуемого поручения, мы можем модифицировать конкретную команду, чтобы задействовать нового получателя команды. Диспетчеру не нужно будет знать, что что-то поменялось, он вызывет метод `execute()` в том же командном объекте, не обращая внимания на факт, что теперь поручением займется совершенно другой конечный исполнитель команды.

Минималистский пример: макрокоманды

Макрокоманды являются очень полезным расширением класса конкретных команд. Они позволяют создавать составные команды, выполняющие целую последовательность простых подкоманд. Вспомните, что происходит, когда вы хотите прекратить выполнение какого-либо приложения. Если в нем есть несохраненные документы, то оно спросит у вас, не хотите ли вы сохранить изменения. Таким образом, команда «выйти» является на самом деле макрокомандой, которая выполняет множество скрытых действий до финального прекращения работы приложения. Эти действия сами являются командами, но мы будем называть их подкомандами при их исполнении в составе макрокоманды.

В классе макрокоманды должно быть имплементировано больше функциональности, чем в простой команде, поскольку им нужно еще определить интерфейс для добавления и удаления подкоманд. Мы расширим исходный интерфейс команды, чтобы он удовлетворял нашим новым требованиям.

Интерфейс макрокоманды

Пример 7.7 содержит в себе интерфейс макрокоманды `IMacroCommand`. Он является расширением интерфейса `ICommand` (из примера 7.1) и дополнительно объявляет методы `add()` и `remove()`.

Пример 7.7. IMacroCommand.as

```
package
{
    public interface IMacroCommand extends ICommand {
        function add(c:ICommand):void;
        function remove(c:ICommand):void;
    }
}
```

Две конкретные подкоманды

Для последующей демонстрации макрокоманды мы создадим два класса конкретных команд (`ConcreteCommand1` и `ConcreteCommand2`), которые используют два класса конечных получателей (`Receiver1` и `Receiver1`). Все они приведены в примерах с 7.8 по 7.11.

Пример 7.8. ConcreteCommand1.as

```
package {
    class ConcreteCommand1 implements ICommand
    {
        var receiver:Receiver1;

        public function ConcreteCommand1(rec:Receiver1):void
        {
            this.receiver = rec;
        }

        public function execute():void
        {
            receiver.action1();
        }
    }
}
```

Пример 7.9. ConcreteCommand2.as

```
package {
    class ConcreteCommand2 implements ICommand
    {
        var receiver:Receiver2;

        public function ConcreteCommand2(rec:Receiver2):void
        {
```

```
            this.receiver = rec;
        }

        public function execute():void
        {
            receiver.action2();
        }
    }
}
```

Пример 7.10. Receiver1.as

```
package {

    class Receiver1 {

        public function action1() {
            trace("Receiver 1: doing action 1");
        }
    }
}
```

Пример 7.11. Receiver2.as

```
package {

    class Receiver2 {

        public function action2() {
            trace("Receiver 2: doing action 2");
        }
    }
}
```

Конкретная макрокоманда

Теперь мы разработаем макрокоманду, которая будет имплементировать интерфейс `IMacroCommand`. Ее имплементация довольно проста, как видно из примера 7.12. Все команды заносятся в массив `commandObjectList` в методе `add()` и потом последовательно выполняются в методе `execute()`.

Пример 7.12. ConcreteMacroCommand.as

```
package
{
    class ConcreteMacroCommand implements IMacroCommand
    {
        var commandObjectList:Array;

        public function ConcreteMacroCommand()
        {
            this.commandObjectList = new Array();
```

```
        }

        public function add(c:ICommand):void
        {
            commandObjectList.push(c);
        }

        public function remove(c:ICommand):void
        {
            for (var i:int = 0; i < commandObjectList.length; i++)
            {
                if (commandObjectList[i] === c)
                {
                    commandObjectList.splice(i, 1);
                    break;
                }
            }
        }

        public function execute():void
        {
            for (var i:int = 0; i < commandObjectList.length; i++)
            {
                commandObjectList[i].execute();
            }
        }
    }
}
```

Объект макрокоманды, созданный клиентом

Сначала клиент создает две подкоманды. Затем он создает новую макрокоманду и добавляет в нее эти две подкоманды. Наконец, он создает диспетчера и задает ему исполнение макрокоманды. Пример 7.13 показывает весь этот процесс.

Пример 7.13. Код клиента по созданию макрокоманды

```
var command1:ICommand = new ConcreteCommand1(new Receiver1());
var command2:ICommand = new ConcreteCommand2(new Receiver2());

// Создаем маркокоманду и добавляем подкоманды
var macroCommand:IMacroCommand = new ConcreteMacroCommand();
macroCommand.add(command1);
macroCommand.add(command2);

var invoker:TimedInvoker = new TimedInvoker();
// устанавливаем макрокоманду в диспетчер
invoker.setCommand(macroCommand);
// вызываем команды по таймеру
invoker.setTimer();
```

Обратите внимание, что макрокоманда не делегирует конечным получателям выполнение требуемого поведения. Ее основной задачей является только исполнение подкоманд. Поскольку она имплементирует интерфейс команды `ICommand`, для диспетчера она неотличима от других командных объектов.

Пример: манипулятор числами

Диспетчер из предыдущего примера мог хранить в себе только один командный объект. Однако в реальных приложениях диспетчеры должны уметь хранить в себе множество команд. Например, посмотрите на меню File (Файл) любого большого приложения. Это хороший пример диспетчера. В таком меню есть позиции меню Open (Открыть), Save (Сохранить) и SaveAs (Сохранить как). Каждая из этих позиций может быть контейнером команды, вызывающим метод `execute()` встроенного командного объекта по «поручению» пользователя. Графические панели инструментов в приложениях – тоже диспетчеры. Они обычно состоят из кнопок с иконками, выполняющих определенные команды по манипуляции элементами в приложении или документе.

В нашем приложении манипулятор числами (рис. 7.4) мы создадим диспетчер, содержащий несколько кнопок, к которым могут быть приписаны определенные командные объекты. При нажатии на такую кнопку будет выполнена соответствующая команда. Наше учебное приложение будет состоять из двух кнопок и одного текстового поля. Эти две кнопки будут иметь встроенные в них командные объекты, которые будут увеличивать или уменьшать числовое значение в текстовом поле.

***Рис. 7.4.** Пример работы манипулятора числами*

Конечно, мы могли бы использовать в нашем приложении встроенный компонент Button («кнопка») из Flash CS3. Однако мы имплементируем собственный такой класс, чтобы показать вам, насколько легко можно создавать произвольные кнопки в ActionScript 3.0. Применение встроенных компонентов будет продемон-

стрировано в главе 12, где мы с вами будем строить приложение с использованием нескольких элементов пользовательского интерфейса как пример реализации шаблона Модель-Представление-Контроллер.

Практичный класс кнопки

Сначала нам нужно создать класс кнопки, который мы будем использовать в последующих наших программах. Пример 7.14 показывает такой класс `TextButton`, являющийся подклассом встроенного в ActionScript 3.0 класса `SimpleButton`. Конструктор нашего класса `TextButton` получает один параметр, который задает текст на данной кнопке. Файл *TextButton.as* содержит встроенный класс `TextButtonState`, представляющий собой подкласс класса `Sprite` и предназначенный для рисования всех требуемых состояний кнопки. Его конструктор получает два параметра: цвет кнопки и ее текст. Он создает новое текстовое поле с полученным текстом и рисует вокруг него заполненный прямоугольник с закругленными углами, используя переданный ему цвет. Создается новый спрайт, который присваивается состояниям `up`, `down` и `over` класса `TextButton`.

Пример 7.14 TextButton.as

```
package
{
    import flash.display.*;
    import flash.events.*;

    public class TextButton extends SimpleButton
    {
        public var selected:Boolean = false;

        public function TextButton(txt:String)
        {
            upState = new TextButtonState(0xFFFFFF, txt);
            downState = new TextButtonState(0x999999, txt);
            overState = new TextButtonState(0xCCCCCC, txt);
            hitTestState = upState;
        }
    }
}

import flash.display.*;
import flash.text.TextFormat;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;

class TextButtonState extends Sprite
{
    public function TextButtonState(color:uint, labelText:String)
```

```
    {
        var label = new TextField();
        label.autoSize = TextFieldAutoSize.LEFT;
        label.text = labelText;
        label.x = 2;
        var format:TextFormat = new TextFormat("Verdana");
        label.setTextFormat(format);
        var buttonWidth:Number = label.textWidth + 10;
        var background:Shape = new Shape();
        background.graphics.beginFill(color);
        background.graphics.lineStyle(2, 0x000000);
        background.graphics.drawRoundRect(0, 0, buttonWidth, 18, 4);
        addChild(background);
        addChild(label);
    }
}
```

Включение триггеров диспетчера кнопкой

Теперь, когда у нас есть кнопки, давайте с их помощью создадим многокнопочного диспетчера. Пример 7.15 содержит класс `InvokerPanel` («панель диспетчера»), который содержит несколько кнопок с присвоенными им командами. В нем определяются два массива `buttonList` и `commandList` для хранения представителей класса кнопки и их соответствующих команд. Его публичный метод `setCommand()` получает два параметра, позицию команды и командный объект (строка 17), и присваивает эту команду на заданную позицию в массиве `commandList`. Метод `setButton()` получает тоже два параметра: позицию кнопки и текст для нее (строка 22). Этот метод создает новую кнопку из класса `TextButton` с заданным текстом и присваивает ей указанное место в массиве `buttonList`. Затем он рисует кнопку, регистрирует ее обработчик событий, реагирующий на щелчки мышью, и добавляет ее в список отображения. Щелчок мышью является триггером для кнопки и ее командного объекта. При щелчке мышью по кнопке событие перехватывается методом `buttonClicked()`, который проходит по списку всех кнопок для определения, какая из них была нажата. После того как такая кнопка найдена, он выполняет команду соответствующего командного объекта из массива `commandList`.

Пример 7.15. InvokerPanel.as

```
 1 package
 2 {
 3     import flash.display.*;
 4     import flash.events.*;
 5
 6     class InvokerPanel extends Sprite
 7     {
 8         var commandList:Array;
 9         var buttonList:Array;
10
```

```
        public function InvokerPanel()
        {
            this.commandList = new Array(5);
            this.buttonList = new Array(5);
        }

        public function setCommand(nSlot:int, c:ICommand):void
        {
            this.commandList[nSlot] = c;
        }

        public function setButton(nSlot:int, sName:String):void
        {
            var btn:TextButton = new TextButton(sName);
            this.buttonList[nSlot] = btn;
            btn.x = nSlot * 100;
            btn.addEventListener(MouseEvent.CLICK, this.buttonClicked);
            this.addChild(btn);
        }

        private function buttonClicked(e:Event)
        {
            for (var i:int = 0; i < buttonList.length; i++)
            {
                if (buttonList[i] === e.target)
                {
                    this.commandList[i].execute();
                    break;
                }
            }
        }
    }
}
```

Команды увеличения и уменьшения

Теперь, когда панель диспетчера `InvokerPanel` готова, мы можем разработать классы команд по увеличению («increment») и уменьшению («decrement») числа в текстовом поле. Примеры 7.16 и 7.17 содержат классы `IncrementCommand` и `DecrementCommand`, которые имплементируют интерфейс команды `ICommand` (пример 7.1). Обратите внимание, что получателем команды является встроенный класс `TextField`, а текст в его поле просто присваивается его свойству `text`. Метод `execute()` получает текстовое значение от получателя, преобразует его в число, выполняет над ним команду и передает полученное значение обратно получателю.

Пример 7.16. IncrementCommand.as

```
package {

    import flash.text.TextField;
```

```
    class IncrementCommand implements ICommand {

        var receiver:TextField;

        public function IncrementCommand(rec:TextField):void {
            this.receiver = rec;
        }

        public function execute():void {
            receiver.text = String(Number(receiver.text) + 1);
        }
    }
}
```

Пример 7.17. DecrementCommand.as

```
package {

    import flash.text.TextField;

    class DecrementCommand implements ICommand {

        var receiver:TextField;

        public function DecrementCommand(rec:TextField):void {
            this.receiver = rec;
        }

        public function execute():void {
            receiver.text = String(Number(receiver.text) - 1);
        }
    }
}
```

Клиент

Нам осталось только разработать код клиента, который создаст командные объекты и присвоит их кнопкам диспетчера. Пример 7.18 показывает нам следующее. Сначала клиент создает получателя команды, являющегося объектом встроенного класса текстового поля (строка 2), и присваивает ему число 100. Потом этот получатель позиционируется на сцене и добавляется к списку отображения (строка 8). Далее клиент создает две конкретные команды увеличения и уменьшения значения получателя (строки 11–12). Затем клиент создает панель кнопок диспетчера с двумя кнопками. И наконец, командные объекты присваиваются нужным позициям для соответствующих кнопок (строки 23–24). Обратите внимание, что позиции кнопок нумеруются от 0 до 4.

Пример 7.18. Код клиента манипулятора числами

```
1  // создаем нового получателя
2  var numDisplayField:TextField = new TextField();
```

```
numDisplayField.autoSize = TextFieldAutoSize.LEFT;
numDisplayField.text = '100'; // начальное значение
numDisplayField.border = true;
numDisplayField.x = 50;
numDisplayField.y = 50;
this.addChild(numDisplayField);

// конкретные командные объекты
var incCommand:ICommand = new  1  IncrementCommand(numDisplayField);
var decCommand:ICommand = new  1  DecrementCommand(numDisplayField);

// создаем панель диспетчера кнопок
var panel:InvokerPanel = new InvokerPanel();
panel.setButton(0,"+1");
panel.setButton(1,"-1");
panel.x = 50;v
panel.y = 100;
this.addChild(panel);

// добавляем команды в диспетчер
panel.setCommand(0, incCommand);
panel.setCommand(1, decCommand);
```

В результате запуска данного примера приложения манипулятора с числами на сцене будет создано текстовое поле с числом 100 и две кнопки с метками «+1» и «–1» (см. рис. 7.4).

Расширенный пример: разделяемые командные объекты

Портативность является очень важной чертой командных объектов. Это их свойство определяется тем, что они содержат в себе все, что только нужно для исполнения конкретной команды. Они не связаны неразрывно (в смысле их классов) ни с получателем, ни с диспетчером команды и удовлетворяют известному, стабильному интерфейсу. Любой сегмент кода в приложении может выполнить с их помощью команду, просто вызвав их метод `execute()`. Чем так хороша их портативность?

Давайте вернемся к нашему примеру с меню File в приложениях. Мы знаем, что это меню можно рассматривать как диспетчер, в котором позиции меню подключены к командным объектам, с помощью которых можно выполнять команды. А как насчет клавиш быстрого вызова для этого меню? Клавиши Ctrl-O на PC и Command-O на Маках выполнят то же действие, что и выбор позиции меню Open. Ctrl-S на PC и Command-S на Маке точно так же сохранят файл, как и выбор позиции меню Save. Значит, клавиши быстрого вызова тоже являются диспетчерами, но нужно ли нам создавать для них новый набор командных объектов? Вовсе нет. Мы можем создать один командный объект и разделить его между многими диспетчерами.

Вызов диспетчера нажатием клавиши

Давайте немного расширим наш пример с манипулятором числами и добавим в него клавиши быстрого вызова для увеличения и уменьшения числа в текстовом поле. Нашим первым шагом является создание нового диспетчера для обработки ввода с клавиатуры. Пример 7.19 содержит такой класс `InvokerKeyboard` («клавиатурный диспетчер»). Структурно он очень похож на нашего предыдущего многокнопочного диспетчера. Однако в отличие от `InvokerPanel` данный класс не должен быть подклассом `Sprite`, поскольку его не нужно добавлять к списку отображения. Объект `Stage` передается классу `InvokerKeyboard`, так как его обработчик событий клавиатуры `onKeyPress()` должен быть зарегистрирован для всей сцены. Это важно для перехватывания всех событий нажатия клавиш.

Два его массива, `keyList` и `commandList`, содержат список клавиш быстрого вызова и список соответствующих им командных объектов. Публичный метод `setCommand()` получает два параметра, код быстрого вызова и командный объект, которые он добавляет тандемом в два вышеназванных массива. Если будут нажаты клавиши, клавиатурный код которых окажется в массиве `keyList`, то будет выполнена соответствующая команда из массива `commandList`.

Пример 7.19. InvokerKeyboard.as

```
package
{
    import flash.events.*;
    import flash.display.Stage;

    class InvokerKeyboard
    {
        var commandList:Array;
        var keyList:Array;

        public function InvokerKeyboard(stageTarget:Stage)
        {
            this.commandList = new Array();
            this.keyList = new Array();
            stageTarget.addEventListener(KeyboardEvent.KEY_DOWN,
                                         this.onKeyPress);
        }

        public function setCommand(keycode:int, c:ICommand):void
        {
            this.keyList.push(keycode);
            this.commandList.push(c);
        }

        private function onKeyPress(event:KeyboardEvent)
        {
            for (var i:int = 0; i < keyList.length; i++)
```

```
                {
                    if (keyList[i] === event.keyCode)
                    {
                        this.commandList[i].execute();
                        break;
                    }
                }
            }
        }
    }
```

Разделение командных объектов клиента

Теперь, когда мы имплементировали клавиатурный диспетчер, мы можем добавить следующий фрагмент кода в конце кода нашего клиента в примере 7.18. Он создает нового представителя класса `InvokerKeyboard` и присваивает ему те же командные объекты, что мы использовали для кнопочного диспетчера.

```
var kb:InvokerKeyboard = new InvokerKeyboard(this.stage);
// добавить команды в клавиатурный диспетчер
kb.setCommand(Keyboard.RIGHT, incCommand);
kb.setCommand(Keyboard.LEFT, decCommand);
kb.setCommand(Keyboard.NUMPAD_ADD, incCommand);
kb.setCommand(Keyboard.NUMPAD_SUBTRACT, decCommand);
```

Теперь «стрелка вправо» на клавиатуре и символ «плюс» на цифровой ее части должны выполнять команду инкремента числа. А «стрелка влево» и символ «минус» на цифровой клавиатуре – команду декремента.

Разделение команд – это мощная особенность шаблона Команда, значительно упрощающая расширение приложений. Например, если мы решим использовать другого получателя команды, то нам надо будет только передать его представителя при создании соответствующего командного объекта. А поскольку все диспетчеры применяют одни и те же командные объекты, такое изменение без проблем распространится по всему приложению. Если бы командные объекты не использовались, а получатели команды вызывались непосредственно из многих диспетчеров, то нам потребовались бы изменения кода во многих местах нашей программы.

Расширенный пример: реализация отмены команд

Другим мощным свойством шаблона Команда является упрощение с его помощью реализации отмены команд, повторения команд, создания очередей команд и ведения их журнала. Мы все хорошо знаем, насколько важно наличие возможности отменить свою команду практически во всех рабочих приложениях, включая игры. Поскольку

командный объект полностью инкапсулирует в себе любую команду, он может так же просто инкапсулировать и обратную к ней команду undo(), возвращающую все к предыдущему состоянию.

Нам нужно расширить интерфейс команды, чтобы определить в нем функцию undo(). Но прежде давайте подумаем, как нам реализовать отмену команд. Для этого нам потребуется хранить стек выполненных команд. Стек – это структура хранения данных, действующая по принципу «первый пришел – первый ушел» (LIFO (last-in-first-out)). В стеке имплементируются две операции – push() («втолкнуть») и pop() («вытолкнуть»), которые соответственно закладывают в него информацию и достают ее оттуда. Операция pop() всегда достает последнюю положенную в стек позицию. Именно она нам и нужна для реализации отмены команд, поскольку она дает последнюю команду. Все команды, которые выполняются, должны закладываться в стек. В идеале на одно приложение должен приходиться один такой стек команд. Если его пользователь захочет вдруг отменить свою последнюю команду, то она должна быть извлечена из стека, и ее функция undo() должна быть выполнена.

Абстрактный интерфейс команд

Вместо того чтобы объявлять обычный интерфейс, мы определим абстрактный интерфейс для команд, поддерживающих свою отмену. Нам это потребуется для имплементации командного стека в классе команд. Пример 7.20 показывает нам абстрактный интерфейс с помощью класса CommandWithUndo («команда с отменой»), который его имплементирует. Обычные массивы в ActionScript 3.0 поддерживают операции push() и pop(), поэтому командный стек будет простым статическим массивом с именем aCommandHistory («история команд»), содержащим в себе все командные объекты, которые уже были выполнены. Имплементацией по умолчанию для метода execute() станет помещение текущего командного объекта в командный стек. Метод undo() будет объявлен нами как абстрактный метод, требующий имплементации в подклассе.

Вспомним, что язык ActionScript 3.0 не поддерживает абстрактные классы. И лишь программист может гарантировать, что абстрактные классы используются только для создания подклассов и все абстрактные методы в них имплементированы, как нужно.

Пример 7.20. CommandWithUndo.as

```
package
{
    // Абстрактный класс (должен быть наследован, а не реализован)
    public class CommandWithUndo implements ICommand
    {
        internal static var aCommandHistory:Array = new Array();
```

```
        public function execute():void
        {
            aCommandHistory.push(this);
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        public function undo():void {}
    }
}
```

Конкретные команды с отменой

Теперь мы имплементируем заново команды увеличения и уменьшения числа, чтобы они удовлетворяли нашему новому интерфейсу, объявленному в абстрактном классе `CommandWithUndo`. Нашими новыми конкретными командами являются `IncrementCommandWithUndo` (пример 7.21) и `DecrementCommandWithUndo` (пример 7.22). Чтобы реализовать возможность отмены команд, нам предварительно надо положить все выполненные командные объекты в командный стек. Это делает метод `execute()` путем вызова в последнем своем выражении такого же метода в суперклассе (строка 17). Затем имплементируется метод `undo()`, который представляет собой просто обратную функцию к методу `execute()` (строка 20).

Пример 7.21. IncrementCommandWithUndo.as

```
 1 package
 2 {
 3    import flash.text.TextField;
 4
 5    class IncrementCommandWithUndo extends CommandWithUndo
 6    {
 7        var receiver:TextField;
 8
 9        public function IncrementCommandWithUndo(rec:TextField):void
10        {
11            this.receiver = rec;
12        }
13
14        override public function execute():void
15        {
16            receiver.text = String(Number(receiver.text) + 1);
17            super.execute();
18        }
19
20        override public function undo():void
21        {
22            receiver.text = String(Number(receiver.text) - 1);
23        }
24    }
25 }
```

Класс DecrementCommandWithUndo аналогичен предыдущему классу и показан в примере 7.22.

Пример 7.22. DecrementCommandWithUndo.as

```
package
{
    import flash.text.TextField;

    class DecrementCommandWithUndo extends CommandWithUndo
    {

        var receiver:TextField;

        public function DecrementCommandWithUndo(rec:TextField):void
        {
            this.receiver = rec;
        }

        override public function execute():void
        {
            receiver.text = String(Number(receiver.text) - 1);
            super.execute();
        }

        override public function undo():void
        {
            receiver.text = String(Number(receiver.text) + 1);
        }
    }
}
```

Кроме того, нам требуется новый командный объект, который будет прикреплен к кнопке отмены команды (с именем Undo) в диспетчере. В примере 7.23 показан класс UndoLastCommand, который будет отменять последнюю выполненную операцию. Его метод execute() первым делом проверяет, содержит ли массив aCommandHistory хоть какие-то командные объекты, и вынимает из него последнюю выполненную команду. Затем он вызывает метод undo() у полученного командного объекта. Обратите внимание, что команда по отмене последней команды не помещает себя в стек. Она вызывает ошибку IllegalOperationError, если будет вызван ее метод undo().

Пример 7.23. UndoLastCommand.as

```
package
{
    import flash.errors.IllegalOperationError;

    class UndoLastCommand extends CommandWithUndo
    {
```

```
        override public function execute():void
        {
            if (aCommandHistory.length)
            {
                var lastCommand:CommandWithUndo = aCommandHistory.pop();
                lastCommand.undo();
            }
        }

        override public function undo():void
        {
            throw new IllegalOperationError("undo operation not supported
                on this command");
        }
    }
}
```

Отменяемые команды, созданные клиентом

В примере 7.24 приведен модифицированный код клиента, который создает командные объекты, используя класс конкретных команд, поддерживающих свою отмену (строки 11–13). На сцену добавляется новая кнопка Undo (строка 19), и к ней прикрепляется соответствующий командный объект (строка 27).

Пример 7.24. Код клиента для манипулятора числами с отменой операций

```
 1 // создаем нового получателя
 2 var numDisplayField:TextField = new TextField();
 3 numDisplayField.autoSize = TextFieldAutoSize.LEFT;
 4 numDisplayField.text = '100'; // начальное значение
 5 numDisplayField.border = true;
 6 numDisplayField.x = 50;
 7 numDisplayField.y = 50;
 8 this.addChild(numDisplayField);
 9
10 // создаем конкретные команды
11 var incCommand:CommandWithUndo = new IncrementCommandWithUndo(numDisplayFie
ld);
12 var decCommand:CommandWithUndo = new DecrementCommandWithUndo(numDisplayFie
ld);
13 var undo:CommandWithUndo = new UndoLastCommand();
14
15// создаем панель диспетчера кнопок
16 var panel:InvokerPanel = new InvokerPanel();
17 panel.setButton(0,"+1");
18 panel.setButton(1,"-1");
19 panel.setButton(2,"Undo");
20 panel.x = 50;
21 panel.y = 100;
22 this.addChild(panel);
```

```
23
24 // добавляем команды в диспетчер
25 panel.setCommand(0, incCommand);
26 panel.setCommand(1, decCommand);
27 panel.setCommand(2, undo);
```

Работа этого учебного приложения будет выглядеть так, как на рис. 7.4, но с добавлением дополнительной кнопки Undo. Команда «повторить операцию», вместе с ведением журнала выполненных операций, может быть реализована похожим способом. Ведение журнала операций очень полезно, когда все выполненные команды должны быть сохранены на диске. Например, запись на диск всех выполненных операций при инсталляции нового продукта позволяет загрузить их оттуда при деинсталляции продукта и последовательно отменить в обратном порядке.

Пример: подкаст-радио

В данном примере приложения мы рассмотрим обычное автомобильное радио с одним новшеством. Вместо того чтобы создавать в нем кнопки для автоматической настройки на определенные радиостанции, мы подключим их к командным объектам, которые будут загружать и проигрывать последний эпизод из подкаста этих радиостанций. Представьте себе, что это такое фантастическое радио из недалекого будущего, когда сети Wi-Fi станут повсеместной реальностью. Вы сможете прослушивать последние новости, когда захотите, не дожидаясь для этого начала следующего часа. Рисунок 7.5 показывает снимок экрана такого приложения. Он состоит из кнопок с названиями радиостанций, на подкаст которых они настроены, и текстового поля, которое показывает название проигрываемого подкаста.

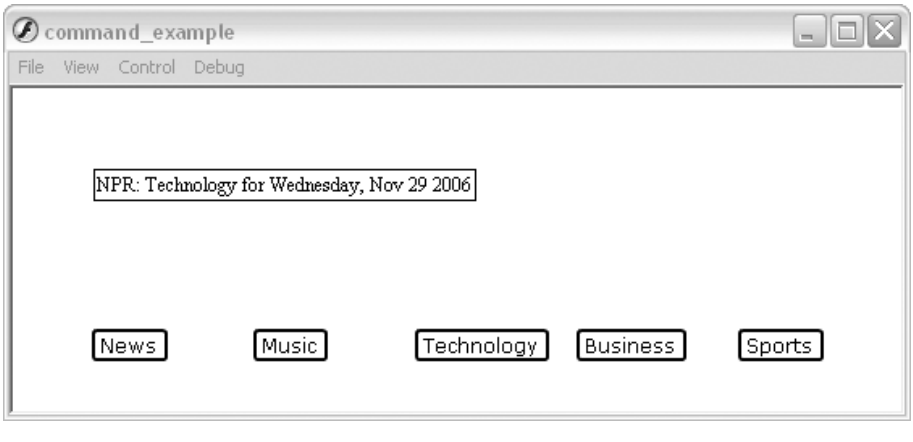


Рис. 7.5. *Снимок экрана подкаст-радио*

Что такое подкаст?

Подкаст – это медиафайл (аудио или видео), распространяемый через Интернет. Подкасты распространяются с помощью специальных новостных лент, которые являются стандартным способом распространения регулярно обновляемого контента в Интернете. Такие новостные ленты являются XML-файлами, как и обычные новостные ленты, где все новости перечисляются от последней к предыдущим. Отличие подкастов от новостных лент состоит в том, что там, где у обычной новости идет текст сообщения, в подкасте стоит адрес URL, указывающий на соответствующий медиафайл. В аудиоподкасте этот файл обычно в формате MP3. В примере 7.25 показан фиктивный подкаст, записанный на XML в формате RSS (из которого многие элементы были для простоты удалены).

Пример 7.25. Фрагмент аудиоподкаста

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
 <channel>
  <title>10AM ET News Summary</title>
  <item>
   <title>News Summary for Saturday, Nov 18 2006 at 10:00 AM EST</title>
   <pubDate>Sat, 18 Nov 2006 10:16:06 EST</pubDate>
   <enclosure url="http://news.podcasts.org/6507084.mp3">
  </item>
 </channel>
</rss>
```

Чтобы прослушать подобный аудиоподкаст, нужно загрузить его XML-файл и извлечь из него атрибут `url` *вложенного* в него элемента, содержащий URL-адрес аудиофайла. Затем этот аудиофайл необходимо загрузить из Интернета и воспроизвести.

Создание пакета с полезными классами

Сначала нам нужно определить два полезных для нас класса, чтобы создать кнопку и текстовые поля, отображаемые на сцене. Первым классом будет уже рассмотренный нами класс `TextButton` из примера 7.14. Мы создадим также класс `TextDisplayField` (подкласс `TextField`), который будет форматировать и отображать название воспроизводимого в данный момент подкаста. Мы поместим оба эти класса в пакет под названием `utils` («полезности»).

Класс `TextDisplayField` показан в примере 7.26. Он довольно прост, и его главными задачами являются: установка начального текста в поле, установка размера шрифта и отображение границы текстового поля.

Пример 7.26. TextDisplayField.as

```
package utils {

    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;

    public class TextDisplayField extends TextField {

        public function TextDisplayField(labelText:String = "",
                                          fontSize:int = 14,
                                          showborder:Boolean = true) {
            autoSize = TextFieldAutoSize.LEFT;
            text = labelText;
            border = showborder;
            var format:TextFormat = new TextFormat("Verdana");
            format.size = fontSize;
            setTextFormat(format);
        }
    }
}
```

Теперь, когда все нужные нам утилитарные классы созданы, мы можем приступить к разработке элементов шаблона Команда для нашего приложения.

Создание команды для подкаста

Нашим интерфейсом команды будет тот же интерфейс `ICommand`, что был определен в примере 7.1. А классом *конкретной команды* будет класс `PlayPodcastCommand`, показанный в примере 7.27. Его конструктор получает два параметра, получатель команды типа `Radio` и URL подкаста типа `String`.

Пример 7.27. PlayPodcastCommand.as

```
package
{
    class PlayPodcastCommand implements ICommand
    {
        var receiver:Radio;
        var podCastURL:String;

        public function PlayPodcastCommand(rec:Radio, url:String):void
        {
            this.receiver = rec;
            this.podCastURL = url;
        }

        public function execute():void
        {
```

```
                this.receiver.playPodcast(this.podCastURL);
            }
        }
    }
```

Разработка класса Radio

Наш класс *получателя* команды с именем `Radio`, являющийся подклассом `Sprite`, показан в примере 7.28. Он использует класс `TextDisplayField` (см. пример 7.26) из ранее разработанного пакета `utils` для отображения текстового поля с названием текущего подкаста (строки 20–21). Ссылка на представителя данного текстового поля хранится в свойстве этого класса `audioDisplay`. Кроме того, в нем определяется статическое свойство `audioChannel` типа `SoundChannel` (строка 15). Данное свойство определено как статическое (`static`), чтобы гарантировать, что только один подкаст будет проигрываться в любой момент времени, даже если в приложении будет несколько представителей класса `Radio`. Метод класса `playPodcast()` загружает XML-файл и регистрирует обработчик события `xmlLoaded()` (строка 28) для перехвата события `Event.COMPLETE`, которое будет сигнализировать об окончании загрузки. После этого полученный XML-файл разбирается с помощью E4X (ECMAScript for XML), входящего в ActionScript 3.0, для получения заголовка первого подкаста (строка 42) и его атрибута содержимого (т. е. ссылки на аудиофайл) (строка 44). Затем аудиофайл подкаста загружается и проигрывается с помощью `audioChannel` (строки 46–51).

Пример 7.28. Radio.as

```
 1 package
 2 {
 3
 4     import flash.display.*;
 5     import flash.events.*;
 6     import flash.media.Sound;
 7     import flash.media.SoundChannel;
 8     import flash.net.*;
 9     import utils.*;
10
11     class Radio extends Sprite
12     {
13
14         private var audioDisplay:TextDisplayField;
15         private static var audioChannel:SoundChannel =  new SoundChannel();
16         var xmlLoader:URLLoader;
17
18         public function Radio()
19         {
20             audioDisplay = new TextDisplayField("click button to play", 14);
21             this.addChild(audioDisplay);
22         }
```

```
23
24          public function playPodcast(url:String)
25          {
26              var xmlURL:URLRequest = new URLRequest(url);
27              this.xmlLoader = new URLLoader(xmlURL);
28              xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
29              xmlLoader.addEventListener(IOErrorEvent.IO_ERROR, loadError);
30          }
31
32          private function xmlLoaded(evtObj:Event)
33          {
34              var xml:XML = new XML();
35              xml = XML(xmlLoader.data);
36              // устанавливаем пространство имен XML по умолчанию
37              if (xml.namespace("") != undefined)
38              {
39                  default xml namespace = xml.namespace("");
40              }
41              // пишем в поле отображения имя звукового потока
42              this.audioDisplay.text = xml..item[0].title;
43              // получаем url звука
44              var url = xml..item[0].enclosure.attribute("url");
45              // загружаем звук и проигрываем
46              var request:URLRequest = new URLRequest(url);
47              var audio:Sound = new Sound();
48              audio.addEventListener(IOErrorEvent.IO_ERROR, loadError);
49              audio.load(request);
50              audioChannel.stop(); // останавливаем играющий звук
51              audioChannel = audio.play();
52          }
53
54          private function loadError(event:Event):void
55          {
56              trace("Load error " + event);
57          }
58      }
59  }
```

Кнопочные диспетчеры для класса Radio

Класс `ControlButton`, показанный в примере 7.29, практически идентичен классу `InvokerPanel` (пример 7.15), который мы обсуждали ранее. Единственным отличием здесь является то, что теперь нам надо импортировать класс `TextButton` из пакета `utils`. Главной задачей данного класса является хранение представителя класса кнопки и связанного с ней командного объекта и выполнение его команды при нажатии кнопки.

Пример 7.29. ControlButtons.as

```
package
{
```

```
import flash.display.*;
import flash.events.*;
import utils.*;

class ControlButtons extends Sprite
{
    var commandList:Array;
    var buttonList:Array;

    public function ControlButtons()
    {
        this.commandList = new Array(5);
        this.buttonList = new Array(5);
    }

    public function setCommand(nSlot:int, c:ICommand):void
    {
        this.commandList[nSlot] = c;
    }

    public function setButton(nSlot:int, sName:String):void
    {
        var btn:TextButton = new TextButton(sName);
        this.buttonList[nSlot] = btn;
        btn.x = nSlot * 100;
        btn.addEventListener(MouseEvent.CLICK,
            this.buttonClicked);
        this.addChild(btn);
    }

    private function buttonClicked(e:Event)
    {
        for (var i:int = 0; i < buttonList.length; i++)
        {
            if (buttonList[i] === e.target)
            {
                this.commandList[i].execute();
                break;
            }
        }
    }
}
}
```

Назначение клиентом подкастов кнопкам

В примере 7.30 клиент сначала создает получателя команды и добавляет его в список отображения (строки 1–5). Затем он создает кнопку, которая представляет у нас диспетчера. Метка на кнопке соответствует названию источника подкаста. Наконец, создается объект конкретной команды и назначается соответствующей кнопке диспетчера (строки 25–29). Кон-

структор класса `PlayPodcastCommand` получает в качестве параметров URL-адрес подкаста и представителя класса получателя команды. Данный код клиента может быть запущен из *класса документа* Flash.

Пример 7.30. Код клиента для подкаст-радио

```
// создаем класс Radio (получатель)
var radio:Radio = new Radio();
radio.x = 50;
radio.y = 50;
this.addChild(radio);

// создаем кнопки (диспетчер)
var controls:ControlButtons = new ControlButtons();
controls.setButton(0,"News");
controls.setButton(1,"Music");
controls.setButton(2,"Technology");
controls.setButton(3,"Business");
controls.setButton(4,"Sports");
controls.x = 50;
controls.y = this.stage.stageHeight - 50;
this.addChild(controls);

// подключаем команды подкаст-станций к кнопкам диспетчера
var podcastURL_1:String = "http://www.npr.org/rss/podcast.php?id=500005";
var podcastURL_2:String = "http://www.npr.org/rss/podcast.php?id=1039";
var podcastURL_3:String = "http://www.npr.org/rss/podcast.php?id=1019";
var podcastURL_4:String = "http://www.npr.org/rss/podcast.php?id=1095";
var podcastURL_5:String = "http://www.npr.org/rss/podcast.php?id=4499275";

controls.setCommand(0, new PlayPodcastCommand(radio, podcastURL_1));
controls.setCommand(1, new PlayPodcastCommand(radio, podcastURL_2));
controls.setCommand(2, new PlayPodcastCommand(radio, podcastURL_3));
controls.setCommand(3, new PlayPodcastCommand(radio, podcastURL_4));
controls.setCommand(4, new PlayPodcastCommand(radio, podcastURL_5));
```

Расширенный пример: динамическое присвоение командного объекта

Вспомним об обычной радиомагнитоле в автомобиле, принимающей станции AM и FM диапазонов. Ее кнопки могут быть по-разному запрограммированы для этих диапазонов. Какая именно радиостанция будет поймана, зависит от того, какой диапазон в данный момент активен. Если вы выберете AM диапазон (нажав на соответствующую кнопку магнитолы), то программируемые кнопки будут настраивать заложенные в них радиостанции AM диапазона. Если же вы выберете FM диапазон, кнопки будут настраивать радиостанции FM диапазона. Таким образом, эти кнопки контекстнозависимые. Хорошим примером команд, доступных в зависимости от контекста, является также *панель*

свойств приложения Flash. Доступные на ней команды меняются в зависимости от типа объекта, выбранного на сцене. И активными остаются только те команды, которые имеют отношение к данному объекту.

Благодаря уже упоминавшейся портативности командных объектов, мы можем присваивать их динамически и заменять прямо во время исполнения программы. Пока что во всех примерах, которые мы рассмотрели, команды присваивались диспетчерам от клиента во время компиляции программы. И если мы присваивали кнопке какую-либо команду, то она оставалась там на все время и не менялась. Теперь мы хотим расширить наш учебный пример с подкаст-радио, чтобы иметь возможность назначать кнопкам командные объекты динамически. Рисунок 7.6 показывает такое расширенное приложение с двумя темами подкастов: музыка и новости. Оно будет работать аналогично нашей AM/FM магнитоле в автомобиле. Командные объекты будут в нем назначаться динамически кнопкам с номерами от 1 до 3. Если будет нажата кнопка «музыка», то при нажатии кнопок 1–3 будут играть музыкальные подкасты. А при нажатой кнопке «новости» эти кнопки будут проигрывать новостные подкасты.

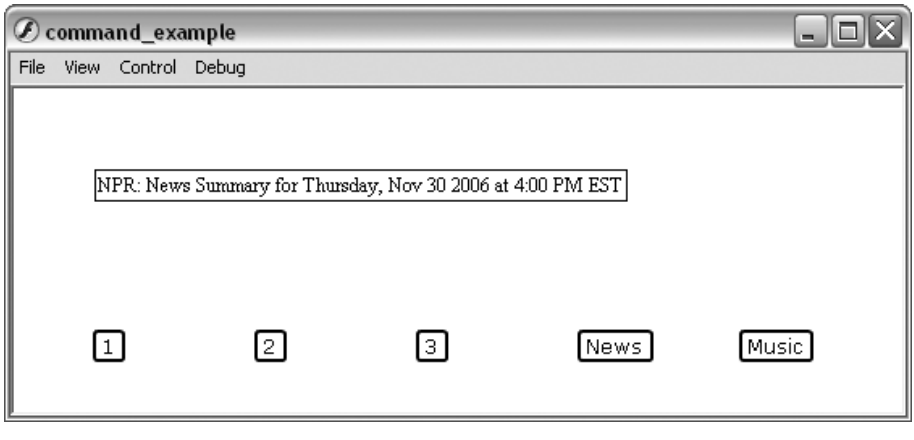


Рис. 7.6. *Подкаст-радио с кнопками тем вещания*

Контекстно-зависимый диспетчер

Чтобы назначать команды динамически в нашем новом приложении, диспетчер должен знать о его статусе. Ему придется назначать различные наборы командных объектов своим кнопкам подкаст-радио в зависимости от состояния приложения или, в нашем случае, от выбранной темы вещания.

Класс `DynamicControlButtons`, показанный в примере 7.31, является расширением класса `ControlButtons` из примера 7.29. Он хранит текущую выбранную тему вещания в свойстве `currentGenre` (строка 7). Два вари-

анта таких тем подкастов определяются статическими константами NEWS и MUSIC (строки 5–6). В нем также определяются и инициализируются два массива (строки 9–10), хранящие в себе командные объекты, присваиваемые трем кнопкам радиостанций при выборе тем вещания новости и музыка.

Метод setGenre() устанавливает одну из тем вещания в свойство currentGenre (строки 18–27). Метод setGenreCommand() (строки 29–38) присваивает переданные ему команды в два массива, содержащие музыкальные и новостные командные объекты. При любых изменениях статуса приложения вызывается метод updateCommandButtons(), чтобы динамически присвоить командные объекты выбранной темы кнопкам радиостанций (в позиции 1–3 массива commandList).

Пример 7.31. DynamicControlButtons.as

```
 1 package
 2 {
 3     class DynamicControlButtons extends ControlButtons
 4     {
 5         public static const NEWS:uint = 0;
 6         public static const MUSIC:uint = 1;
 7         var currentGenre:uint = NEWS;
 8
 9         var newsPodcastCommands:Array;
10         var musicPodcastCommands:Array;
11
12         public function DynamicControlButtons()
13         {
14             this.newsPodcastCommands = new Array(3);
15             this.musicPodcastCommands = new Array(3);
16         }
17
18         public function setGenre(genre:uint)
19         {
20             if (genre == NEWS)
21             {
22                 this.currentGenre = NEWS;
23             } else if (genre == MUSIC) {
24                 this.currentGenre = MUSIC;
25             }
26             this.updateCommandButtons();
27         }
28
29         public function setGenreCommand(nSlot:int, c:ICommand, genre:uint):void
30         {
31             if (genre == NEWS)
32             {
33                 this.newsPodcastCommands[nSlot] = c;
34             } else if (genre == MUSIC) {
```

```
35                 this.musicPodcastCommands[nSlot] = c;
36             }
37             this.updateCommandButtons();
38         }
39
40         private function updateCommandButtons()
41         {
42             for (var i:int = 0; i < 3; i++)
43             {
44                 if (currentGenre == NEWS)
45                 {
46                     this.commandList[i] = this.newsPodcastCommands[i];
47                 } else if (currentGenre == MUSIC) {
48                     this.commandList[i] =  this.musicPodcastCommands[i];
49                 }
50             }
51         }
52     }
53 }
```

Команды динамического присвоения командных объектов

Чтобы динамически присваивать командные объекты, нам нужно создать две конкретные команды по установке темы вещания – музыка либо новости. Это будут классы `SetToMusicGenreCommand` (пример 7.32) и `SetToNewsGenreCommand` (пример 7.33).

Пример 7.32. SetToMusicGenreCommand.as

```
package
{
    class SetToMusicGenreCommand implements ICommand
    {
        var receiver:DynamicControlButtons;

        public function SetToMusicGenreCommand(
            rec:ControlButtons)[1]:void
        {
            this.receiver = rec;
        }

        public function execute():void
        {
            this.receiver.setGenre(
```

[1] Здесь тип параметра `rec` должен быть `DynamicControlButtons`, иначе при компиляции будет выдана ошибка несоответствия типов (родительский класс не может присваиваться переменной, типизированной как его подкласс). – *Прим. науч. ред.*

```
                DynamicControlButtons.MUSIC);
        }
    }
}
```

Пример 7.33. SetToNewsGenreCommand.as

```
package
{
    class SetToNewsGenreCommand implements ICommand
    {
        var receiver:DynamicControlButtons;

        public function SetToNewsGenreCommand(
            rec:ControlButtons[1]):void
        {
            this.receiver = rec;
        }

        public function execute():void
        {
            this.receiver.setGenre(DynamicControlButtons.NEWS);
        }
    }
}
```

Обратите внимание, что получателем команд в обоих случаях является диспетчер `DynamicControlButtons`. Здесь диспетчер представляет собой также исполнителя команд, устанавливающих тему вещания.

Установка динамического назначения команд со стороны клиента

Клиент должен специфицировать командные объекты трем кнопкам радиостанций для обеих тем вещания и команды по смене этих тем (последние две кнопки). Само динамическое переназначение командных объектов кнопкам в зависимости от темы вещания происходит в диспетчере. Клиент, по сути, программирует кнопки в своем радиоприложении практически так же, как кнопки на своей магнитоле в автомобиле.

Сначала клиент создает *получателя* и добавляет его в список отображения. Потом он создает *диспетчера*, присваивает названия всем его пяти кнопкам и добавляет его в список отображения. Переменным присваиваются URL-адреса подкастов (три URL-адреса для каждой темы вещания). Далее клиент выполняет очень важную работу по созданию командных объектов класса `PlayPodcastCommand` и присваиванию их кнопкам радиостанций в каждой из тем вещания. Наконец, клиент соз-

[2] Аналогичное замечание. – *Прим. науч. ред.*

дает и присваивает командные объекты выбора тем соответствующим кнопкам диспетчера. Пример 7.34 показывает весь этот процесс.

Пример 7.34. Код клиента по настройке расширенного подкаст-радио

```
// создаем класс Radio (получатель)
var radio:Radio = new Radio();
radio.x = 50;
radio.y = 50;
this.addChild(radio);

// создаем кнопки (диспетчер)
var controls:DynamicControlButtons = new DynamicControlButtons();
controls.setButton(0,"1");
controls.setButton(1,"2");
controls.setButton(2,"3");
controls.setButton(3,"News");
controls.setButton(4,"Music");
controls.x = 50;
controls.y = this.stage.stageHeight - 50;
this.addChild(controls);

// ссылки на подкасты
var podcastNewsURL_1:String =
    "http://www.npr.org/rss/podcast.php?id=500005";
var podcastNewsURL_2:String =
    "http://rss.cnn.com/services/podcasting/newscast/rss.xml";
var podcastNewsURL_3:String =
    "http://www.npr.org/rss/podcast.php?id=510053";
var podcastMusicURL_1:String =
    "http://www.npr.org/rss/podcast.php?id=510019";
var podcastMusicURL_2:String =
    "http://www.npr.org/rss/podcast.php?id=510026";
var podcastMusicURL_3:String =
    "http://minnesota.publicradio.org/tools/podcasts/
    new_classical_tracks.xml";

// добавляем команды станций на кнопки
controls.setGenreCommand(0, new PlayPodcastCommand(radio,
    podcastNewsURL_1), DynamicControlButtons.NEWS);
controls.setGenreCommand(1, new PlayPodcastCommand(radio,
    podcastNewsURL_2), DynamicControlButtons.NEWS);
controls.setGenreCommand(2, new PlayPodcastCommand(radio,
    podcastNewsURL_3), DynamicControlButtons.NEWS);
controls.setGenreCommand(0, new PlayPodcastCommand(radio,
    podcastMusicURL_1), DynamicControlButtons.MUSIC);
controls.setGenreCommand(1, new PlayPodcastCommand(radio,
    podcastMusicURL_2), DynamicControlButtons.MUSIC);
controls.setGenreCommand(2, new PlayPodcastCommand(radio,
    podcastMusicURL_3), DynamicControlButtons.MUSIC);
```

```
// добавляем команды выбора темы вещания
controls.setCommand(3, new SetToNewsGenreCommand(controls));
controls.setCommand(4, new SetToMusicGenreCommand(controls));
```

Заключение

Шаблон Команда – это очень мощный пример использования инкапсуляции, или сокрытия информации; он применим в очень многих стандартных ситуациях проектирования программного обеспечения.

В сущности шаблон Команда заключает требуемое поведение в командный объект. Все команды выполняются с помощью вызова метода `execute()` в этом командном объекте. Каким классам делегировано выполнение скрытых в командных объектах поручений и какими методами эти классы его имплементируют, совершенно не известно тому, кто вызывает указанный метод. Это отделяет код, вызывающий исполнение поручения, от кода, который его имплементирует.

Такое разделение делает командные объекты портативными, что позволяет применять их во многих ситуациях. Один командный объект может использоваться несколькими диспетчерами и в разных частях кода приложения. Благодаря этому легко менять или расширять поведение всего приложения.

Одним из самых полезных свойств командных объектов является то, что они могут быть переданы диспетчеру во время выполнения программы. Данное свойство дает возможность изменять поведение элементов приложения в зависимости от его текущего статуса, что позволяет делать приложения контекстнозависимыми.

И, кроме того, шаблон Команда позволяет имплементировать в приложении такие стандартные возможности, как цепочки команд (или макрокоманды), отмена и повторение команд, а также ведение их журнала.

8

Шаблон Наблюдатель

Именно теория определяет,
что можно наблюдать на практике.

Альберт Эйнштейн

Каждый человек, который зорко наблюдает
и стойко принимает решения,
неосознанно вырастает в гения.

Эдвард Дж. Бульвер-Литтон
(Автор бессмертной строки «Это была темная ветреная ночь», написанной при попытке объяснить, почему у него пропал Интернет, и он не выполнил свой проект в срок.)

Мир полон очевидных вещей, которые никто не замечает.

Шерлок Холмс (Оригинальное описание синтаксических ошибок.)

Можно многое увидеть, просто наблюдая.

Йоги Берра[1]

Что такое шаблон Наблюдатель?

На понятийном уровне объяснить смысл шаблона проектирования Наблюдатель очень легко. Его главной задачей является отсылка информации своим подписчикам. Он работает, как газета или кабельное теле-

[1] Американский баскетболист. Известен своими «глубокомысленными» замечаниями, вроде «Нельзя думать и бить одновременно». – *Прим. перев.*

видение. Когда кто-то на них подписывается, его обслуживание начинается и продолжается вплоть до прекращения подписки.

В приложениях, в которых информацию из единственного источника надо передавать нескольким различным приемникам, применение одного передатчика имеет больше смысла, чем получение информации с помощью повторяющихся запросов к источнику данных. Например, в случае использования веб-сервиса, постоянно выдающего цену акций, эффективнее создать в приложении один приемник этих данных, который будет рассылать ее всем нуждающимся в ней, чем осуществлять запросы от всех них к указанному веб-сервису. Если ваше приложение принимает данные о цене акций и выводит эту информацию в виде таблиц и различных диаграмм, наличие многочисленных подписок на каждый из классов форматирования потребовало бы отдельных, повторяющихся запросов к веб-сервису. Использование только одного получателя данных о цене акций, который будет передавать их потом всем остальным, позволяет вам существенно снизить объем ваших запросов к веб-сервису.

Кроме повышения эффективности работы единственный источник данных гарантирует, что все его подписчики получат одну и ту же информацию. А если все они самостоятельно получают информацию, то вообразите, что произойдет, если между их запросами произойдет значительное обновление высылаемых данных. Первый подписчик делает запрос к сервису и форматирует получаемую информацию в виде таблицы, а второй запрашивает у сервиса данные, чтобы построить на их основе диаграмму. Данные в диаграмме и в таблице не будут соответствовать друг другу, хотя так не должно быть. При использовании шаблона Наблюдатель в результате одного запроса к веб-сервису всем подписчикам *будут разосланы данные из одного и того же запроса*. Поэтому данные для таблицы и для диаграммы всегда будут одинаковыми.

В приложениях, в которых происходят изменения высокоуровневых данных, шаблон Наблюдатель помогает существенно сократить использование системных ресурсов. Все данные отправляются одному источнику, из которого они рассылаются всем подписчикам. Например, в боевых играх счет постоянно изменяется вследствие постоянно происходящих событий в игре. Если какой-то объект в результате боевых действий оказывается «уничтоженным», ему больше не нужно получать информацию, и он должен перестать принимать ее. Если же какие-то объекты входят в игру или «воскрешаются», они должны начать получать данные. Все это может стать кошмаром для программиста, если у него не будет некой системы, которая будет собирать всю информацию об измененных данных и распределять ее различным элементам игры, одновременно отслеживая изменения в подписках. В такой ситуации шаблон Наблюдатель является просто спасением, поскольку решает все вопросы по сбору и распределению данных.

Основные характеристики

Главной характеристикой шаблона Наблюдатель является то, что изменение состояния происходит в одном месте, а информация об этом рассылается всем подписанным на него элементам. Такое отношение одинко-многим позволяет программистам создавать слабосвязанные классы и к тому же поддерживать целостность информации. Рисунок 8.1 демонстрирует схему отношений между исходным источником данных, классом рассылки и классами наблюдателей.

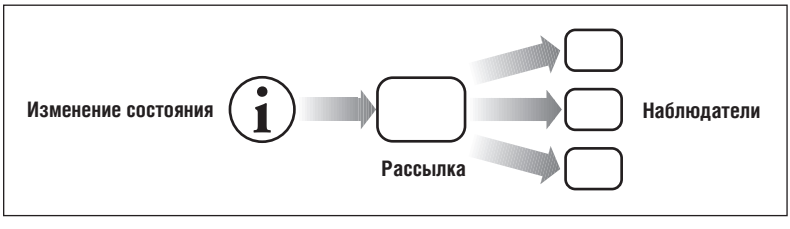


Рис. 8.1. *Потоки информации в шаблоне Наблюдатель*

Рассылка информации классам подобным образом не только эффективна, но и допускает их расширение. Например, данные по цене акций могут рассылаться в различные статистические модели, инкапсулированные в различные классы. При появлении новой статистической модели добавление ее в подобное приложение так же просто, как создание очередного такого класса и подключение его к подписке на данную тему рассылки.

Можно перечислить следующие основные характеристики шаблона Наблюдатель:

- Целостность данных для слабосвязанных, но взаимодействующих объектов.
- Получатели данных (наблюдатели) могут подписываться и отписываться на получение данных.
- Один источник (рассылка) рассылает информацию об изменении состояния.
- Любое количество наблюдателей может быть подписано на рассылку.

Чтобы понять важность целостности и непротиворечивости информации, представьте себе заседание ООН. Предположим, что представитель Китая произносит речь о новой торговой политике своей страны. Эта информация должна быть переведена с китайского на все другие языки, представленные в ООН. Хороший шаблон Наблюдатель в данной ситуации посылает данную речь от одного источника (рассылки) всем переводчикам (наблюдателям), чтобы они перевели ее на языки слушателей. Если бы наблюдатели получали указанную речь из разных источников,

то шансы на нарушение целостности информации сильно бы возросли; и если бы неверные данные были получены, то новая торговая политика Китая могла бы быть неверно истолкована.

Главными элементами шаблона Наблюдатель являются *рассылка* и *наблюдатель*. Рассылка *уведомляет* наблюдателя о любом изменении состояния. В этом шаблоне применяется *композиция*, а не наследование в качестве основной техники для повторного использования. При этом рассылка делегирует операции наблюдателю. Данный процесс будет подробно объясняться далее в этой главе, в разделе «Ключевые концепции ООП, используемые в шаблоне Наблюдатель».

Модель шаблона Наблюдателя

Диаграмма классов шаблона Наблюдатель ясно показывает его два главных элемента: рассылку и наблюдателя. Интерфейс рассылки определяет методы для оповещения наблюдателей об изменении состояния. Рассылка содержит методы подписки и отписки. Возможность подписывать объекты к рассылке позволяет легко добавлять их в список на получение новых данных. Весь набор методов рассылки передается для использования наблюдателю. А самому наблюдателю нужен только метод по получению обновления статуса. Конкретный класс рассылки применяет этот метод в наблюдателе в своем методе нотификации, определенном в интерфейсе рассылки. Кроме того, конкретная рассылка может иметь у себя методы получатель и установщик для нового состояния. Конкретный класс наблюдателя должен имплементировать свой метод `update()`. В нем тоже могут быть методы подписки и отписки, причем скорее первый из них, а отписка может предоставляться конкретным объектам рассылки либо происходить при определенных состояниях. Рисунок 8.2 показывает диаграмму классов шаблона Наблюдатель.

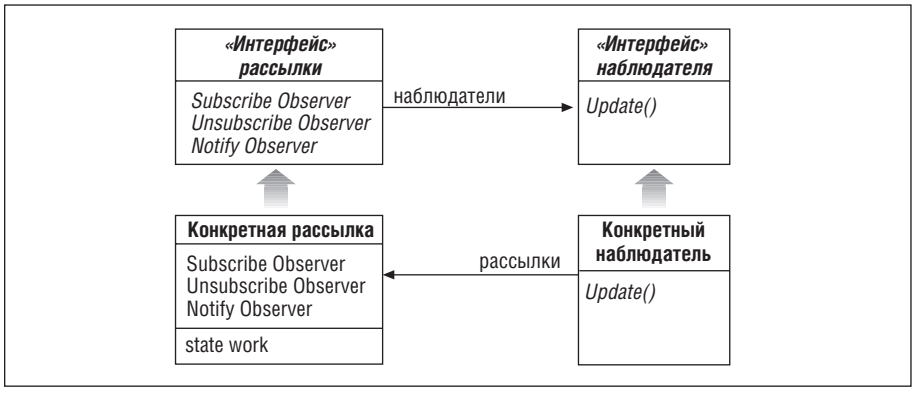


Рис. 8.2. *Диаграмма классов шаблона Наблюдатель*

Мы можем разбить эту диаграмму на четыре части: рассылка, наблюдатель, конкретная рассылка, конкретный наблюдатель. Интерфейс рассылки `Subject` устанавливает основные информационные связи между ней и наблюдателем с помощью методов подключения и отключения наблюдателя от процесса нотификации о новых данных. Фактически *каждый* метод рассылки предназначен для наблюдателя. А интерфейс наблюдателя `Observer`, с единственным методом `update()`, предназначен для процесса нотификации о новых данных со стороны рассылки. Конкретный класс рассылки `ConcreteSubject` обеспечивает свойство `stateWork` для хранения состояния, о котором были уведомлены наблюдатели. Он также ответственен за нотификацию о новых данных подписанных наблюдателей. И наконец, с помощью метода `update()` конкретный класс наблюдателя `ConcreteSubject` поддерживает текущее значение состояния согласованным с рассылкой, а с помощью свойства `state` он хранит его у себя.

Ключевые концепции ООП, используемые в шаблоне Наблюдатель

Одним из принципов объектно-ориентированного проектирования, выдвинутым среди прочих Гаммой (Erich Gamma) и его коллегами в их классической книге по шаблонам проектирования, является *предпочтение композиции объектов наследованию классов*. Он означает, что при разработке программы следует по возможности применять композицию объектов, а не строить иерархическую структуру, начиная с некоторого суперкласса и доводя до конца с помощью наследования. Чтобы понять причины такой рекомендации, необходимо внимательнее посмотреть на *композицию объектов*.

А чтобы оценить преимущества композиции перед наследованием, нам нужно понять другой принцип, выдвинутый Гаммой (Erich Gamma), Хелмом (Richard Helm), Джонсоном (Ralph Johnson) и Влиссидесом (John Vlissides):

> Программируйте, используя интерфейсы, а не их имплементации.

Давайте сначала рассмотрим именно этот принцип.

Используйте интерфейс

В двух словах, согласно Гамме и его коллегам программирование с использованием интерфейсов лучше, чем использование их имплементации, поскольку это снижает зависимости в имплементациях подсистем. И, следовательно, увеличивает гибкость программного обеспечения. В результате некоторые изменения в программе могут быть сделаны прямо во время ее работы.

Идея применения при программировании интерфейсов не подразумевает буквального использования в ActionScript 3.0 конструкции с ключевым словом `interface` вместо конструкций с ключевым словом `class`. Вы можете использовать в программе интерфейс абстрактного класса или обычный интерфейс вместо классов. Так что же именно она означает?

Легче всего понять эту идею, если думать о методах абстрактного класса и интерфейса как об *интерфейсах*. Интерфейс – это набор функций, имплементированных в объекте. Эрик и Элизабет Фриман (Eric Freeman, Elisabeth Freeman) (авторы книги «Head First Design Patterns» (издательство O'Reilly, 2004)) предложили представлять свойства абстрактного класса или интерфейса как *супертипы*. Оказывается, на практике существует множество подходов к использованию интерфейсов в программе, но чтобы понять его основу, давайте рассмотрим следующие скрипты в примерах с 8.1 по 8.6. В примерах с 8.1 по 8.4 определяется интерфейс, а в примерах 8.5 и 8.6 он имплементируется. Сохраните все эти файлы в одном каталоге.

Пример 8.1. SpaceWarrior.as («космический воин»)

```
//SpaceWarrior.as
package
{
    interface SpaceWarrior
    {
        function useWeapon():void;
    }
}
```

Пример 8.2. Alien.as («пришелец»)

```
//Alien.as
package
{
    public class Alien implements SpaceWarrior
    {
        function Alien()
        {
            // Конструктор
        }
        public function useWeapon():void
        {
            trace("Zaaaapp!!!");
        }
    }
}
```

Пример 8.3. Earthling.as («землянин»)

```
//Earthling.as
package
```

```
{
    public class Earthling implements SpaceWarrior
    {
        function Earthling()
        {
            // Конструктор
        }
        public function useWeapon():void
        {
            trace("Ka Boom!!!");
        }
    }
}
```

Пример 8.4. Main.as

```
//Main.as
package
{
    import flash.display.Sprite;
    public class Main extends Sprite
    {
        public function Main()
        {
            // Программирование через имплементацию
            var alien:Alien=new Alien();
            alien.useWeapon();

            // Программирование через интерфейс
            var spaceWarrior:SpaceWarrior=new Earthling();
            spaceWarrior.useWeapon();
        }
    }
}
```

Откройте новый файл документа Flash и в его поле класс документа введите `Main`. Когда вы протестируете это приложение, вы увидите:

```
Zaaaapp!!!
Ka Boom!!!
```

Первое выражение `trace()` выводит то, что возвращается, когда `Alien` вызывает метод `useWeapon()`, а второе – то, что возвращается, когда `Earthling` вызывает этот метод. Чтобы понять, в чем состоит разница между программированием при использовании имплементации и программированием при использовании интерфейсов, посмотрите на следующий код из скрипта *Main.as*.

```
// Программирование через имплементацию
var alien:Alien=new Alien();
alien.useWeapon();
```

```
// Программирование через интерфейс
var spaceWarrior:SpaceWarrior=new Earthling();
spaceWarrior.useWeapon();
```

В первой его части используется имплементация, поскольку объект alien объявлен (типизирован) с помощью имплементации Alien (класс Alien имплементирует интерфейс SpaceWarrior). Во второй части данного фрагмента кода переменная spaceWarrior определяется супертипом SpaceWarrior. Но создается она с помощью конструктора Earthling(). Как видите, все это прекрасно работает. Более того, мы можем переопределить spaceWarrior как объект Alien, и код по-прежнему будет работать. Замените последнюю часть данного фрагмента следующим кодом:

```
// Программирование через интерфейс
var spaceWarrior:SpaceWarrior=new Earthling();
spaceWarrior.useWeapon();
spaceWarrior=new Alien();
spaceWarrior.useWeapon();
```

Теперь на выходе мы увидим:

```
Zaaaapp!!!
Ka Boom!!!
Zaaaapp!!!
```

Как видите, при программировании с использованием интерфейса переменные получаются гораздо более гибкими. Нам не нужно знать, что заключается в самой функции (интерфейсе) и что в нем будет делаться. Но мы знаем, что он будут выполнять то, что нужно нам, в зависимости от конструктора, который мы применяем.

Композиция объектов

Чтобы понять, почему композиция объектов предпочтительнее наследования классов, нам нужно рассмотреть, что они делают. В предыдущих главах мы уже использовали композицию и наследование, и у вас, наверно, уже сложилось некоторое представление о них. Обе эти концепции были представлены нами в главе 1, но поскольку шаблон Наблюдатель наглядно иллюстрирует использование композиции, мы рассмотрим ее здесь еще раз.

Вспомнив о том, что сравнение между композицией объектов и наследованием классов происходит с точки зрения построения гибких и готовых к повторному использованию элементов, мы можем переписать наш принцип следующим образом:

> Для повышения гибкости и готовности к повторному использованию элементов программ лучше использовать при их создании композицию объектов, чем наследование классов.

Использование разработанного с применением наследования кода довольно очевидно. Разработчики берут существующий класс и исполь-

зуют его заново вместе с новыми свойствами в создаваемых подклассах. Поскольку внутренние свойства классов обычно видимы для их подклассов, такое их применение в имплементации называют *использованием белого ящика*.

Композиция объектов, напротив, добавляет новую функциональность при сведении вместе существующих объектов. В результате композиции класс использует функциональность другого класса, тогда как наследование дает зависимость от класса, обладающего функциональностью. Таким образом, при композиции один класс имеет отношение «использует» с функциональностью другого класса, а при наследовании класс имеет отношение «является» с классом, имеющим функциональность, унаследованную от другого класса. Поскольку композиция объектов скрывает внутренние детали, разработку с ее применением называют *использованием черного ящика*.

Преимуществами наследования являются легкость его разработки, его наглядность и относительная простота внесения некоторых изменений. Например, если требуется, чтобы изменения произошли во всех подклассах, вам нужно всего лишь изменить их суперкласс; все изменения наследуются потомками.

Самым большим недостатком наследования считается *нарушение инкапсуляции*. Оно происходит, когда подкласс изменяется при изменении его родительского класса. Такое изменение может нарушить функциональность в данном подклассе. (Инкапсуляция может быть нарушена и другими способами и действиями, не связанными с наследованием, но они не имеют отношения к рассматриваемой теме.) Другим недостатком наследования является наследование нежелательных свойств. Подклассы обязаны перенять от родительского класса полный набор его качеств; как мы видели в главе 4, для решения данной проблемы используется замещение. Наконец, поскольку наследование осуществляется во время компиляции, вы не можете изменить вашу имплементацию изменением родительского класса во время исполнения программы. Все эти вышеуказанные недостатки снижают гибкость программ.

Одно из преимуществ композиции объектов противоположно главному недостатку наследования – композиция сохраняет инкапсуляцию. Композиция объектов занимается только тем, что каждый из ее объектов делает и в каких отношениях он находится с другими объектами. Такая сосредоточенность каждого класса (объекта) на простых и ограниченных задачах позволяет композиции лучше и проще сохранять инкапсуляцию. Центр внимания смещается здесь с вопроса, как что-то работает, к взаимоотношениям между объектами, а поскольку каждый из объектов получается в чем-то зависимым от других, вопросы их инкапсуляции и надежной работы становятся самыми главными.

При наличии множества надежно инкапсулированных и сосредоточенных на своих задачах объектов вам, вероятно, придется создавать боль-

ше классов для реализации требуемой функциональности, и этот факт считается недостатком композиции объектов. Однако такие узкоспециализированные классы предполагают небольшие иерархии, и композиция дает возможность одному объекту *пользоваться* свойствами другого объекта, вместо того чтобы наследовать свойства и *быть* другим объектом.

Шаблон проектирования Наблюдатель использует композицию объектов, а не наследование. Его двумя главными интерфейсами и конкретными классами являются *рассылка* и *наблюдатель*. Вместе эти объекты составляют единую *композицию*, которая создает общую модель проекта приложения.

Минималистский абстрактный Наблюдатель

Чтобы создать работоспособное приложение, реализующее шаблон Наблюдатель, нам нужны как минимум два интерфейса и два класса согласно модели из диаграммы классов (см. рис. 8.2):

- Интерфейс рассылки
- Интерфейс наблюдателя
- Конкретный класс рассылки
- Конкретный класс наблюдателя

Принимая во внимание нашу основную цель создать централизованное распределение информации об изменении состояния, мы займемся реализацией методов подписки и отписки от рассылки, метода уведомления об изменении состояния и метода обновления состояния для наблюдателя. Чтобы они правильно работали вместе, мы будем использовать массив для хранения информации обо всех наблюдателях-подписчиках и метод дистрибуции уведомлений об изменении текущего состояния.

Интерфейс рассылки

В интерфейсе рассылки нам нужны только три метода, которые решат все вопросы с подпиской и позаботятся об уведомлении. Поскольку все функции в конструкции интерфейса являются абстрактными, нам не придется заниматься всякими мелкими деталями. Однако нам необходимо аккуратно проследить за тем, чтобы присутствовали все нужные нам части. Пример 8.5 показывает нам сценарий, который нужно сохранить в файле *Subject.as*.

Пример 8.5. Subject.as

```
package
{
    // Интерфейс рассылки
    public interface Subject
    {
```

```
        function subscribeObserver(o:Observer):void;
        function unsubscribeObserver(o:Observer):void;
        function notifyObserver():void;
    }
}
```

Уже в первых двух функциях видны признаки *композиции*. Параметр `o` объявлен через тип данных `Observer` («наблюдатель»), ссылающийся на другой интерфейс, который будет построен как часть шаблона проектирования Наблюдатель. Если вы посмотрите на диаграмму классов на рис. 8.2, вы увидите стрелку от абстрактного класса `Subject` («рассылка») к абстрактному классу `Observer`. Такая ссылка на тип данных наблюдателя является частью процесса, использующего композицию, а не наследование.

Интерфейс наблюдателя

Интерфейс наблюдателя `Observer` выглядит обманчиво просто. Его единственная функция `update()` является на самом деле частью композиции между структурами `Observer` и `Subject`. Однако мы сможем увидеть эту композицию, только когда перейдем к рассмотрению связи между конкретным наблюдателем и рассылкой через метод супертипа `update()` в интерфейсе `Observer`. Сохраните код из примера 8.6 как *Observer.as*.

Пример 8.6. Observer.as

```
package
{
    // Интерфейс наблюдателя
    public interface Observer
    {
        function update(light:String):void;
    }
}
```

Чтобы проиллюстрировать работу шаблона проектирования Наблюдатель в его минимальной форме, мы выбрали пример с освещением в комнате, которое может быть только в состоянии «включено» или «выключено». Хотя в абстрактном интерфейсе мы просто указали, что типом данных будет строка. То есть ее значением могли бы быть и некоторые промежуточные состояния освещения. У нас появилась некоторая гибкость в приложении. Наша цель – простота и ясность нашего примера, но в принципе функция `update()` из шаблона Наблюдатель может иметь дело с множеством своих параметров самых разнообразных типов. Позднее в данной главе мы это продемонстрируем.

Конкретный класс рассылки

На данной стадии нам нужно заставить наш интерфейс работать. Для этого в конкретном классе рассылки `ConcreteSubject` необходимо сделать следующее:

- Создать массив для хранения подписчиков-наблюдателей.
- Создать свойство для хранения состояния как часть данного класса.
- Реализовать все детали процесса подписки, связанные с супертипом `Observer` и массивом наблюдателей.
- Разработать процесс удаления (прекращения подписки) элементов массива наблюдателей с использованием супертипа `Observer` для его параметров.
- Создать процесс для уведомлений (нотификации), связанный с методом `update()` из интерфейса `Observer`.

Для процесса нотификации нам потребуется некоторый метод-установщик. Эта функция будет связана с типом данных, меняющих состояние, о котором будут уведомляться все наблюдатели. В свою очередь, при изменении состояния (вызове функции установщика) должна срабатывать и функция уведомления. Сохраните сценарий из примера 8.7 в файле *ConcreteSubject.as*.

Пример 8.7. ConcreteSubject.as

```
package
{
    public class ConcreteSubject implements Subject
    {
        private var light:String;
        private var observers:Array;
        function ConcreteSubject ()
        {
            trace ("*|*Concrete Subject*|*");
            observers=new Array();
        }
        public function subscribeObserver (obserNow:Observer):void
        {
            observers.push (obserNow);
        }
        public function unsubscribeObserver (obserNow:Observer):void
        {
            for (var ob:int=0; ob<observers.length; ob++)
            {
                if (observers[ob]==obserNow)
                {
                    observers.splice (ob,1);
                    break;
                }
            }
        }
        public function notifyObserver ():void
        {
            for (var notify in observers)
            {
```

```
                observers[notify].update (light);
                trace ("Observer " + notify + " Light is "+light);
            }
        }
        public function setLight (light:String):void
        {
            this.light=light;
            notifyObserver ();
        }
    }
}
```

В этой имплементации основная работа осуществляется с помощью композиции. Задача конструктора состоит в создании только одной конструкции: реализации массива. (Выражения `trace()` являются избыточными и добавлены нами только для того, чтобы вы могли яснее видеть работу структуры наблюдателя.) Функции подписки и отказа от подписки используют композицию с объектом супертипа `Observer` в качестве своего параметра. Функция нотификации применяет метод `update()` из интерфейса `Observer`. Таким образом, здесь вы видите работу композиции, а не объектов подкласса.

Когда вы будете тестировать данное приложение, вы будете использовать интерфейсы вместо их имплементаций. Чтобы это было возможно, проверьте следующую строчку:

```
public function unsubscribeObserver(obserNow:Observer):void
```

Какой бы объект наблюдателя ни был помещен в параметр `obserNow`, последний определяется через свой супертип (`Observer`), а не свой тип имплементации. (В разделе «Работа шаблона Наблюдатель» процесс использования интерфейса объясняется подробнее.)

Конкретный класс наблюдателя

Конкретный класс наблюдателя `ConcreteObserver` должен реализовать гораздо более узкую задачу, чем класс `ConcreteSubject`. Он сохраняет состояние, за которым он наблюдает, чтобы обеспечить требуемую целостность данных у всех наблюдателей-подписчиков. Сохраните сценарий из примера 8.8 как *ConcreteObserver.as*.

Пример 8.8. ConcreteObserver.as

```
package
{
    // Конкретный наблюдатель
    class ConcreteObserver implements Observer
    {
        private var light:String;
        function ConcreteObserver()
        {
```

```
            trace("=Concrete Observer=");
        }
        public function update(light:String):void
        {
            this.light=light;
        }
    }
}
```

Здесь функция `update()` использует класс `ConcreteSubject` для сохранения своего параметра. Таким образом, все объекты класса `ConcreteObserver` оказываются привязанными к одному источнику обновления информации, которое будет выполнять функция нотификации конкретной рассылки.

Работа шаблона Наблюдатель

Теперь мы можем посмотреть, как все наши части будут работать вместе. В сценарии, имплементирующем наше приложение, нам нужно создать объекты класса `ConcreteSubject` и класса `ConcreteObserver`. Затем нам необходимо подписать наблюдателей на получение новостей от объекта `ConcreteSubject`. Кроме того, просто чтобы убедиться, что все работает правильно, нам следует отписать от рассылки по крайней мере одного наблюдателя. Для проверки всей функциональности нам надо изменить наблюдаемое нами состояние с помощью функции `setLight()` на другое значение. При этом только подписанные к рассылке наблюдатели должны наблюдать такие изменения. Таким образом, сначала подписав к рассылке наблюдателей, а затем отписав от рассылки по крайней мере одного из них, мы должны убедиться, что все работает так, как ожидалось. Пример 8.9 имплементирует шаблон проектирования Наблюдатель. Сохраните его код как *TestSub.as*.

Пример 8.9. TestSub.as

```
package
{
    import flash.display.Sprite;
    public class TestSub extends Sprite
    {
        public function TestSub()
        {
            var mySub:ConcreteSubject=new ConcreteSubject();
            var subObserver:Observer=new ConcreteObserver();
            var subObserver2:Observer=new ConcreteObserver();
            var subObserver3:Observer=new ConcreteObserver();
            mySub.subscribeObserver(subObserver);
            // subObserver передается как реализация супертипа Observer
            mySub.subscribeObserver(subObserver2);
            mySub.subscribeObserver(subObserver3);
            mySub.setLight("on");
```

```
                mySub.unsubscribeObserver(subObserver);
                mySub.setLight("off");
            }
        }
    }
```

Обратите внимание, что данный сценарий использует интерфейс для создания представителей различных наблюдателей, но должен применять напрямую класс конкретной рассылки `ConcreteSubject` для ее имплементации. Однако мы могли бы объявить наших наблюдателей и с помощью типа `ConcreteObserver`, а потом использовать их как объекты типа `Observer` в качестве параметров для подписки. В любом случае мы бы следовали рекомендации использовать при программировании интерфейсы.

Откройте новый документ Flash, в его поле класс документа введите `TestSub` и протестируйте наше приложение. На выходе вы должны увидеть следующее:

```
*|*Concrete Subject*|*
=Concrete Observer=
=Concrete Observer=
=Concrete Observer=
Observer 0 Light is on
Observer 1 Light is on
Observer 2 Light is on
Observer 0 Light is off
Observer 1 Light is off
```

Как вы можете видеть по результатам работы выражений `trace()`, один объект воспользовался конструктором класса `ConcreteSubject`, и три – конструктором класса `ConcreteObserver`. Наблюдаемое состояние было вначале установлено в `on` («включено»), и, как и ожидалось, все три наблюдателя восприняли его правильно. Потом один из наблюдателей был отключен от подписки, и состояние было изменено на `off` («выключено»). Только два оставшихся наблюдателя показали прием этого состояния.

Пример: множество состояний и идентификация пользователей

Минималистский пример реализации шаблона проектирования Наблюдатель использовал только одно изменяющееся состояние. Возможно, вы думаете, что добавлять в него дополнительные переменные состояния сложно, но вы увидите, что на самом деле это очень просто. Одной из главных черт шаблонов проектирования является их гибкость, и добавление новых состояний прекрасно ее иллюстрирует.

В некоторых приложениях, использующих шаблон Наблюдатель, может быть важна точная идентификация наблюдателей. В нашем минималистском примере в предыдущем разделе наблюдатели различались

по их индексу в массиве, который изменялся при удалении наблюдателя. Однако реализация идентификации наблюдателей по свойству также довольно проста.

Множественные состояния

Чтобы понять множественные состояния, представьте себе ежедневную газету, имеющую несколько постоянных разделов. Их список может содержать, например, следующие темы, нуждающиеся в регулярном обновлении:

- Новости
- Спорт
- Акции
- Развлечения

Все эти темы могут быть представлены с помощью строкового типа данных, но для разнообразия мы сделаем данные об акциях числовым типом. В нашем минималистском примере свойство состояния появлялось первым делом при объявлении параметра метода `update()`. Если мы хотим иметь дополнительные свойства состояний, нам нужно лишь добавить больше свойств. Таким образом, в интерфейсе `Observer` нам необходимо изменить функцию `update()` следующим образом:

```
function update(news:String,sports:String,stocks:Number,entertainment:
    String):void;
```

Это очень просто. Как видите, наличие различных типов данных не создает никаких проблем.

Конкретной рассылке нужно хранить в себе все состояния, поэтому каждому состоянию необходима собственная переменная для хранения текущего состояния. Функция-установщик также теперь должна работать со всеми четырьмя состояниями, поэтому она тоже должна быть расширена.

```
public function setType(news:String,sports:String,stocks:Number,
    entertainment:String)
{
    this.news=news;
    this.sports=sports;
    this.stocks=stocks;
    this.entertainment=entertainment;
    notifyObserver();
}
```

Обратите внимание, функция `setType()` структурно идентична функции `setLight()` в примере со светом. Изменилось только число параметров функции и количество переменных для их хранения.

Наконец, в конкретной рассылке нам нужно поменять число параметров в функции update(), рассылающей информацию об изменении состояний:

```
this.observers[notify].update(news,sports,stocks,entertainment);
```

И опять вы можете видеть, что изменилось только число параметров, но не ее структура.

В классе конкретного наблюдателя все изменения также связаны с числом переменных, а не с его структурой. Теперь их стало четыре вместо одной. А главная функция update() изменилась следующим образом:

```
public function update(news:String,sports:String,stocks:Number,
    entertainment:String):void
{
    this.news=news;
    this.sports=sports;
    this.stocks=stocks;
    this.entertainment=entertainment;
}
```

Ее структура, как и других элементов шаблона Наблюдатель, осталась прежней. Существенному изменению в приложении подлежит лишь формат вывода информации. Структура выводимых данных не поменяется, однако, имея четыре различных состояния, вам придется так отформатировать вывод, чтобы в нем присутствовали все различные состояния (категории информации) и они были отделены друг от друга.

Кто вы такой?

В некоторых приложениях вам потребуется знать, кто является вашим подписчиком. Возможно, вам даже понадобится гарантия того, что один и тот же человек случайно не подписался к вам два раза, особенно когда у всех подписчиков имеются уникальные имена.

Чтобы все это реализовать, нам нужно начать с конкретного класса ConcreteObserver. Используя строковую переменную, мы дадим имена каждому из наших подписчиков.

```
// Идентификатор подписчика
internal var subName:String;

// Конструктор
function ConcreteObserver(subName:String):void
{
    trace(subName + " has subscribed");
    this.subName=subName;
}
```

Добавив имя пользователя к его конструктору и связав это имя с создаваемым объектом, мы позволили каждому наблюдателю иметь собствен-

ное имя. Теперь класс ConcreteObserver имеет свойство subName, а каждый элемент в массиве observers может использовать данное свойство для самоидентификации.

Чтобы гарантировать, что только один наблюдатель с заданным именем может подписаться к рассылке, функция подписки ее конкретного класса ConcreteSubject должна быть изменена. Поместив в нее условное выражение, которое будет проверять, что имена наблюдателей не совпадают (что один наблюдатель не подписался дважды к одной рассылке), мы сможем отклонять подписку пользователя с повторяющимся именем.

```
// Добавляем переменную для проверки дублирования
private var duplicate:Boolean;


....(код)

// Подписаться и предотвратить повторную подписку
public function subscribeObserver(obserNow:Observer):void
{

    duplicate=false;
    for(var ob=0;ob<this.observers.length;ob++)
    {
        if(this.observers[ob]==obserNow)
        {
            duplicate=true;
            trace( this.observers[ob].subName+ " already a subscriber.
                \n");
        }
    }
    if(! duplicate)
    {
        this.observers.push(obserNow);
    }
}
```

В некоторых случаях вам, возможно, не потребуется ограничивать наблюдателей с одним именем только одной подпиской. Чтобы позволить подписываться нескольким наблюдателям с одним именем, просто не добавляйте запрещающий это код, приведенный выше, в вашу программу.

Обновленная реализация Наблюдателя

Теперь наше приложение, реализующее шаблон Наблюдатель, имеет гораздо больше функциональности. Не только каждый из наблюдателей идентифицируется по собственному имени, но и запрещается подписка к рассылке более одного наблюдателя с одним и тем же именем. Кроме того, теперь у нас четыре отслеживаемых состояния с уведомлением об

их изменении. Примеры с 8.10 по 8.13 следует сохранить в одном каталоге под именами, указанными в заголовках примеров.

Пример 8.10. Subject.as

```
package
{
    // Интерфейс рассылки
    public interface Subject
    {
        function subscribeObserver(o:Observer):void;
        function unsubscribeObserver(o:Observer):void;
        function notifyObserver():void;
    }
}
```

Пример 8.11. Observer.as

```
package
{
    // Интерфейс наблюдателя
    public interface Observer
    {
    function update(news:String,sports:String,stocks:Number,entertainment:
        String):void;
    }
}
```

Пример 8.12. ConcreteSubject.as

```
package
{
    // Конкретная рассылка
    public class ConcreteSubject implements Subject
    {
        private var news:String;
        private var sports:String;
        private var stocks:Number;
        private var entertainment:String;
        private var observers:Array;

        // Добавляем переменную для проверки дубликатов
        private var duplicate:Boolean;

        // Конструктор
        public function ConcreteSubject ():void
        {
            trace ("*|*Concrete Subject*|*");
            observers=new Array();
        }
        // Подписка и предотвращение повторной подписки
        public function subscribeObserver (obserNow:Observer):void
        {
```

```
    var duplicate:Boolean = false;
    for (var ob=0; ob<observers.length; ob++)
    {
        if (observers[ob]==obserNow)
        {
            duplicate=true;
            trace (observers[ob].subName+ "
                is already a subscriber.\n");
        }
    }
    if (! duplicate)
    {
        observers.push (obserNow);
    }
}
// Отписываемся и убираем из массива
public function unsubscribeObserver (obserNow:Observer):void
{
    for (var ob=0; ob<observers.length; ob++)
    {
        if (observers[ob]==obserNow)
        {
            observers.splice (ob,1);
        }
    }
}
// Создаем уведомление и задаем формат вывода
public function notifyObserver ():void
{
    for (var notify in this.observers)
    {
        observers[notify].update (news,sports,
            stocks,entertainment);
        var nowNews:String=" sees that "+
            news + " is interesting,";
        var nowSports:String = " and learns that " + sports;
        var nowStocks:String=".\nWhoaa!,
            the stock market is at " + stocks;
        var nowEntertain:String=" and "+ entertainment +
            " is showing at the Bijou.";
        trace (observers[notify].subName + nowNews +
            nowSports + nowStocks + nowEntertain);
    }
}
// Добавляем все необходимые состояния
public function setType (news:String,sports:String,stocks:Number,
    entertainment:String):void
{
    this.news=news;
    this.sports=sports;
    this.stocks=stocks;
```

```
            this.entertainment=entertainment;
            notifyObserver ();
        }
    }
}
```

Пример 8.13. ConcreteObserver.as

```
package
{
    // Конкретный наблюдатель
    class ConcreteObserver implements Observer
    {
        // Храним дополнительные состояния
        private var news:String;
        private var sports:String;
        private var stocks:Number;
        private var entertainment:String;

        // Идентификатор подписчика
        public var subName:String;

        // Конструктор
        function ConcreteObserver(subName:String):void
        {
            trace(subName + " has subscribed");
            this.subName=subName;
        }

        // Добавляем состояния в функцию обновления
        public function update(news:String,sports:String,stocks:Number,
            entertainment:String):void
        {
            this.news=news;
            this.sports=sports;
            this.stocks=stocks;
            this.entertainment=entertainment;
        }
    }
}
```

Если хотите, вы можете изменить текст в различных «заголовках» нашей газеты «The Daily Bugle» (Ежедневный Горн).

Издание «Горна»

Итак, мы реализовали шаблон проектирования Наблюдатель, симулирующий подписку на газету, которую мы назвали «The Daily Bugle». Теперь пора протестировать ее процессы подписки и распространения новостей. Наш тестовый класс клиента подписки должен иметь всего несколько элементов:

- Конкретную рассылку
- Подписчиков (конкретных наблюдателей)
- Ежедневные новости (изменения состояния)
- Отказ от подписки (удаление из списка наблюдателей)

На этот раз нам потребуются имена для всех наблюдателей, и мы ожидаем гораздо больше информации на выходе приложения, но в целом его структура существенно не изменится. Пример 8.14 содержит весь необходимый код нашего теста, который нужно сохранить как *BugleSubscribe.as*.

Пример 8.14. BugleSubscribe.as

```
package
{
    // Приложение, тестирующее Наблюдателя
    import flash.display.Sprite;
    public class BugleSubscribe extends Sprite
    {
        public function BugleSubscribe()
        {
            var bigNews:ConcreteSubject=new ConcreteSubject();
            var larry:ConcreteObserver=new ConcreteObserver("Larry");
            var mo:ConcreteObserver=new ConcreteObserver("Mo");
            var curly:ConcreteObserver=new ConcreteObserver("Curly");
            var shemp:ConcreteObserver=new ConcreteObserver("Shemp");
            bigNews.subscribeObserver(larry);
            bigNews.subscribeObserver(mo);
            bigNews.subscribeObserver(curly);
            bigNews.subscribeObserver(shemp);

            // Устанавливаем состояние 1
            bigNews.setType("Computer Invented","Home Team Wins",
                3234.54," Mad Duck");

            // Отключаем 2-х наблюдателей и пробуем
               снова подписать текущего подписчика
            trace("\n** Larry and Shemp have unsubscribed **\n");
            bigNews.unsubscribeObserver(larry);
            bigNews.unsubscribeObserver(shemp);

            // Пробуем подписаться снова
            bigNews.subscribeObserver(mo);

            // Устанавливаем состояние 2
            bigNews.setType("Memory Prices Down","Game Rained Out",
                2987.98," Bad Bug");
        }
    }
}
```

Проследите, чтобы файл *BugleSubscribe.as* был сохранен в том же каталоге, что и другие файлы классов. Теперь создайте новый документ Flash и введите `BugleSubscribe` в его поле свойств класс документа. Когда вы протестируете приложение, вы должны увидеть следующий результат его работы:

```
*|*Concrete Subject*|*
Larry has subscribed
Mo has subscribed
Curly has subscribed
Shemp has subscribed
Larry sees that Faster Computer Invented is interesting, and learns
    that Home Team Wins.
Whoaa!, the stock market is at 3234.54 and The Mad Duck is showing
    at the Bijou.
Mo sees that Faster Computer Invented is interesting, and learns
    that Home Team Wins.
Whoaa!, the stock market is at 3234.54 and The Mad Duck is showing
    at the Bijou.
Curly sees that Faster Computer Invented is interesting, and learns
    that Home Team Wins.
Whoaa!, the stock market is at 3234.54 and The Mad Duck is showing
    at the Bijou.
Shemp sees that Faster Computer Invented is interesting, and learns
    that Home Team Wins.
Whoaa!, the stock market is at 3234.54 and The Mad Duck is showing
    at the Bijou.

** Larry and Shemp have unsubscribed **

Mo is already a subscriber.

Mo sees that Memory Prices Down is interesting, and learns that Game
    Rained Out.
Whoaa!, the stock market is at 2987.98 and The Bad Bug is showing
    at the Bijou.
Curly sees that Memory Prices Down is interesting, and learns that Game
    Rained Out.
Whoaa!, the stock market is at 2987.98 and The Bad Bug is showing
    at the Bijou.
```

Когда происходит каждая новая подписка, выводятся имя нового наблюдателя и сообщение о факте его подписки. Все четыре подписчика видят одинаковый формат данных, поскольку он заложен в класс `ConcreteSubject`. Однако это достаточно легко изменить, просто убрав все форматирование и оставив рассылку необработанных данных. Шаблон проектирования Наблюдатель следит за тем, чтобы информация об изменении состояния, рассылаемая подписавшимся наблюдателям, была одинаковой, но не сильно беспокоится о ее форматировании. Это должны быть просто одни и те же данные.

Применяя метод unsubscribeObserver, можно удалить выбранного наблюдателя из листа подписки – с помощью строки вида:

```
bigNews.unsubscribeObserver(ConcreteObserver instance);
```

Как вы видели, наблюдатели shemp и larry были удалены из подписки между первой и второй рассылками новостей с использованием этого метода.

Динамическое изменение состояний

До этого момента тестирование шаблона Наблюдатель происходило путем неторопливого присвоения новых значений состоянию с помощью его методов-установщиков; наша задача состояла в том, чтобы посмотреть, как работает структура этого шаблона. Однако в более реальных приложениях, использующих Flash и ActionScript 3.0, состояния меняются очень быстро, и уведомления об их изменениях должны посылаться практически постоянно.

Чтобы мы посмотрели, как быстро меняющиеся данные могут быть отосланы к единственному источнику и затем разосланы многочисленным наблюдателям, нашим следующим приложением будет динамичная игра с тремя противниками. Два из них будут представлять собой космические корабли, пускающие ракеты и космические торпеды, а третий – космическую станцию, стреляющую по кораблям боевыми лучами. В этой игре немногое можно сделать – только пострелять из космических кораблей и космической станции и понаблюдать, как разрушаются два космических корабля. Однако она служит иллюстрацией того, как все попадания генерируются событиями на сцене и рассылаются различным противникам. Когда один из сражающихся выбывает из игры, он перестает быть подписчиком такой рассылки, и все, что от него остается, – это его последнее сообщение перед разрушением.

Запись космического сражения

В данном примере приложения на основе шаблона Наблюдатель интерфейсы рассылки Subject и наблюдателя Observer содержат только числовые и строковые параметры в функциях оповещения наблюдателя и обновления состояния. Функции подписки и отказа от нее такие же, как и в предыдущем примере реализации шаблона Наблюдатель в этой главе. Сохраните классы из примеров 8.15 и 8.16 в файлах с именами, указанными в их заголовках.

Пример 8.15. Subject.as

```
package
{
    // Интерфейс рассылки
    public interface Subject
```

```
    {
        function subscribeObserver(o:Observer):void;
        function unsubscribeObserver(o:Observer):void;
        function notifyObserver(score:Number,damage:String):void;
    }
}
```

Пример 8.16. Observer.as

```
package
{
    // Интерфейс наблюдателя
    public interface Observer
    {
        function update(score:Number,damage:String):void;
    }
}
```

Новые обязанности конкретных классов

Два класса, представляющие конкретные рассылки и наблюдателей, будут немного отличаться от своих аналогов в предыдущем примере. Класс `ConcreteSubject` будет по-прежнему следить за подпиской и отказом от нее, а также слать уведомления об изменении состояния.

А у класса `ConcreteObserver` появится новая обязанность. При создании нового представителя этого класса мы будем автоматически подписывать его к рассылке уведомлений. Таким образом, вместо того, чтобы осуществлять двухэтапный процесс, на первом этапе создавать представителя класса наблюдателя, а на втором подписывать его, класс `ConcreteObserver` будет теперь автоматически подписывать на рассылку свои новые объекты. Для этого нужно добавить всего одну строчку в его конструктор:

```
concreteObserver.subscribeObserver(this);
```

Параметр данного метода указывает на объект, который создается этим конструктором. В любой момент данный наблюдатель может отказаться от подписки и возобновить ее с помощью вызова соответствующих функций. Примеры 8.17 и 8.18 нужно сохранить под именами, указанными в их заголовках.

Пример 8.17. ConcreteSubject.as

```
package
{
    // Конкретная рассылка
    public class ConcreteSubject implements Subject
    {
        private var score:Number;
        private var damage:String;
        private var duplicate:Boolean;
```

```
private var observers:Array;

public function ConcreteSubject ()
{
    observers=new Array();
}
// Подключить наблюдателя без дубликатов
public function subscribeObserver (obserNow:Observer):void
{
    duplicate=false;

    for (var ob=0; ob < observers.length; ob++)
    {
        if (observers[ob] == obserNow)
        {
            duplicate=true;
            trace ("Sorry, " + observers[ob].nomDeGuerre +
                " is already subscribed.");
        }
    }
    if (! duplicate)
    {
        observers.push (obserNow);
    }
}
// Отключить наблюдателя
public function unsubscribeObserver (obserNow:Observer):void
{
    for (var ob=0; ob < this.observers.length; ob++)
    {
        if (observers[ob] == obserNow)
        {
            trace ("\n***" + this.observers[ob].nomDeGuerre +
                " has been removed.***\n");
            observers.splice (ob,1);
        }
    }
}
// Уведомить наблюдателей о счете и текущем уровне повреждений
public function notifyObserver (score:Number,damage:String):void
{
    for (var notify in observers)
    {
        observers[notify].update (score,damage);
    }
}
// Установить счет и повреждения
public function setScore (score:Number,damage:String):void
{
    this.score=score;
    this.damage=damage;
```

```
            notifyObserver (score,damage);
        }
    }
}
```

Пример 8.18. ConcreteObserver.as

```
package
{
    // Конкретный наблюдатель
    class ConcreteObserver implements Observer
    {
        public var nomDeGuerre:String;
        private var damage:String;
        private var score:Number;
        private var concreteObserver:Subject;

        function ConcreteObserver(concreteObserver:Subject)
        {
            this.concreteObserver=concreteObserver;
            concreteObserver.subscribeObserver(this);
        }

        // Информация для вывода
        public function passOn():String
        {
            return "Current score: "+score+"\nCurrent damage: "+ damage;
        }

        // Запомнить состояние
        public function update(score:Number,damage:String):void
        {
            this.score=score;
            this.damage=damage;
            passOn();
        }
    }
}
```

Теперь, когда все базовые программы разработаны, начинается настоящая работа. В следующем разделе вы создадите несколько клипов, представляющих космические корабли и станцию вместе с их вооружением. Однако наиболее важно то, что действия различных элементов будут порождать изменения состояния программы, которые будут обрабатываться с помощью шаблона проектирования Наблюдатель и рассылаться всем враждующим сторонам.

Начало космической битвы

В примере 8.19 вы найдете класс, который устраивает небольшую битву в космосе. Однако, прежде чем начинать работу над классом, вам необходимо создать несколько клипов, а для этого нужно открыть новый

файл документа Flash. Следующие шаги помогут вам подготовить сцену для сражения.

Подготовка документа Flash

1. Откройте новый документ Flash.
2. Установите размер сцены 650×450 и цвет фона серым.
3. Добавьте звезды на свой собственный вкус, используя обычные желтые звезды и падающие звезды. Не слишком много, только чтобы создать ощущение звездного неба.
4. В свойствах документа в поле класс документа введите `ChangeHandler`.
5. Сохраните файл под именем *AliensAttack.fla* в том же каталоге, что и другие файлы данного приложения.

Постройка космических кораблей

После того как ваш документ Flash подготовлен, можно заняться созданием клипов космических кораблей.

1. Выберете из меню Insert → New Symbol.

Когда вы даете имя клипу, который вы собираетесь использовать как класс, обязательно следуйте общепринятым правилам именования классов. Первая буква должна быть большой, и в имени не должны присутствовать пробелы. Так что используйте имя `Earthling`, а не `earthling` и не вставляйте в имена пробелы и другие недопустимые символы.

2. Когда откроется окно New Symbol, введите в поле Name `Earthling` и выберите Movie clip в качестве типа поведения. Если вы видите кнопку с именем Advanced, то нажмите на нее; она откроет новую часть окна New Symbol, которая потребуется нам далее.
3. Отметьте позицию Export for ActionScript в группе Linkage. Вы должны увидеть, что позиция Export in first frame выбрана автоматически. Кроме того, в поле класс вы должны увидеть `Earthling` и в поле базовый класс `flash.display.MovieClip`. Щелкните на OK.
4. Теперь вы в режиме редактирования символа. Центр клипа обозначен перекрестием линий в середине сцены.
5. Используя инструменты рисования, изобразите простейший космический корабль шириной в 83 и высотой в 47 пикселов. Рисунок 8.3 показывает вам пример такого корабля, увеличенного в 4 раза. Когда вы закончите рисовать, кликните на иконке Scene 1, расположенной вверху сцены; вы должны увидеть ваш клип на панели Library.
6. Повторите шаги с 1 по 5, чтобы создать клип под названием `Alien`. Нарисуйте другой корабль, направленный в противоположную сторону по сравнению с кораблем землянина. Когда вы закончите, вы должны увидеть оба клипа на панели Library.

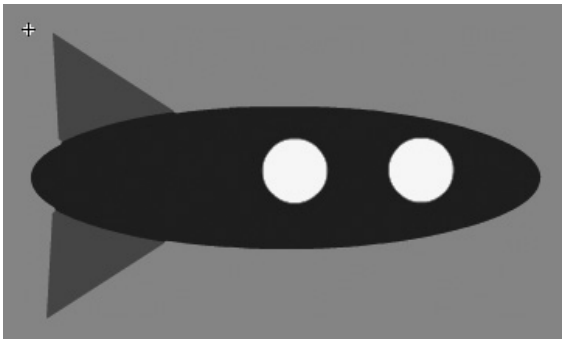


***Рис. 8.3.** Оружие землянина*

Создание вооружений

Изначально мы пытались создать космические корабли вместе с их вооружениями на одном клипе, но после нескольких экспериментов мы убедились, что лучше сделать им различные клипы для большей гибкости. Наши вооружения очень примитивные, но вы можете при желании добавить в них все что угодно: имитацию взрыва, звуки и запуск при нажатии. Следующие шаги позволяют создать их простые формы для демонстрации того, как состояние в шаблоне Наблюдатель может быть изменено с помощью триггеров.

1. Выберете из меню Insert → New Symbol. Назовите ваш клип `bullet` («пуля»), используя для имени буквы нижнего регистра. Выберите для него тип поведения Movie clip и щелкните на OK. Теперь вы в режиме редактирования символа.
2. Нарисуйте что-нибудь, что, по вашему мнению, можно запустить в пришельца. Мы использовали овал со стабилизатором наверху для акулоподобной ракеты со следующими размерами: ширина – 23, высота (включая стабилизатор) – 7.3. Получилась довольно маленькая ракетка, но пропорциональная кораблю землянина. Когда вы закончите рисовать, кликните на иконке Scene 1, чтобы выйти из режима редактирования символа.
3. Выберете из меню Insert → New Symbol. Когда откроется окно New Symbol, введите `Missile` («ракета») в поле Name и выберите для него тип поведения Movie clip. Если вы видите кнопку с именем Advanced, то нажмите на нее; она откроет новую часть окна New Symbol, которая нам потребуется на следующем шаге.
4. Отметьте позицию Export for ActionScript в группе Linkage. Вы должны увидеть, что позиция Export in first frame выбрана автоматически. Кроме того, в поле класс вы должны увидеть `Missile`, а в поле базовый класс – `flash.display.MovieClip`. Щелкните на OK.

5. В режиме редактирования символа перетащите копию клипа bullet на центр, в позицию X=0, Y=0.
6. Щелкните на фрейме 20 и нажмите F5, чтобы добавить фреймы до него. Теперь нажмите F6 и добавьте ключевой фрейм. Выберите ключевой фрейм 20 и передвиньте bullet в позицию X=380, Y=0.
7. Щелкните на фрейме 1 и на панели свойств выберите меню Motion in the Tween. Вы должны увидеть стрелку на временной линии от фрейма 1 до фрейма 20 на синем фоне. При тестировании клип bullet должен двигаться слева направо.
8. Щелкните на фрейме 1 и откройте панель Actions (нажмите F9 или Options + F9 на Маке). Введите stop().
9. Повторите шаги с 1 по 8 для создания оружия пришельца. Пусть его «пуля» выглядит иначе, и ее клип называется projectile. Используйте следующие размеры: ширина – 42, высота – 11. Вместо названия оружия Missile используйте имя Torpedo. Торпеда пришельца должна двигаться в противоположном направлении по сравнению с ракетой землянина. От начальной позиции X=0, Y=0 на первом фрейме она должна попасть в X=–423 на последнем фрейме своего пути. Рисунок 8.4 показывает парочку простых увеличенных примеров нашего оружия.

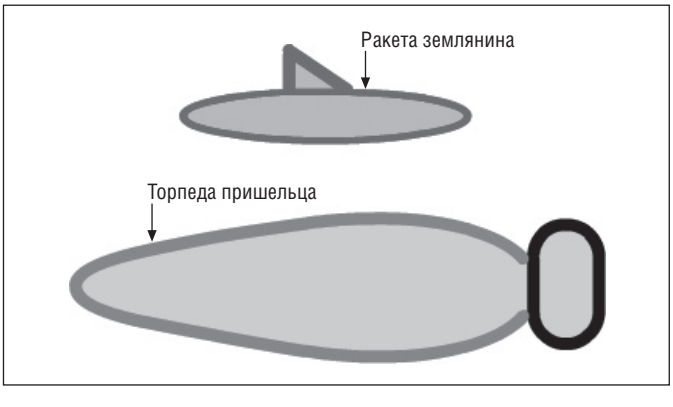


***Рис. 8.4.** Оружие землянина и пришельца*

Создание космической станции андроидов с лучевым оружием

Космическая станция андроидов представляет собой автоматическую боевую станцию, стреляющую двойным боевым лучом по кораблям землянина и пришельца. Это простой пятиугольный объект. Ее V-образный боевой луч находится прямо в ее клипе pentagon.

1. Выберете из меню Insert → New Symbol.

2. Когда откроется окно New Symbol, введите в поле Name Android и выберите Movie clip в качестве типа поведения. Если вы видите кнопку с именем Advanced, то нажмите на нее; она откроет новую часть окна New Symbol, которая потребуется нам далее.
3. Отметьте позицию Export for ActionScript в группе Linkage. Вы должны увидеть, что позиция Export in first frame выбрана автоматически. Кроме того, в поле класс вы должны увидеть Android, а в поле базовый класс – flash.display.MovieClip. Щелкните на OK.
4. Теперь вы в режиме редактирования символа. Используя инструменты рисования, изобразите пятиугольник с шириной в 57 и высотой в 54 пикселов. Когда вы закончите рисовать, кликните на иконке Scene 1, расположенной вверху сцены; вы должны увидеть ваш клип космической станции на панели Library.
5. Выберете из меню Insert → New Symbol.
6. Когда откроется окно New Symbol, введите в поле Name Beam и выберите Movie clip в качестве типа поведения. Если вы видите кнопку с именем Advanced, то нажмите на нее; она откроет новую часть окна New Symbol, которая нам потребуется на следующем шаге.
7. Отметьте позицию Export for ActionScript в группе Linkage. Вы должны увидеть, что позиция Export in first frame выбрана автоматически. Кроме того, в поле класс вы должны увидеть Beam, а в поле базовый класс – flash.display.MovieClip. Щелкните на OK.
8. Теперь вы в режиме редактирования символа. Добавьте еще два уровня в дополнение к существующему. Назовите верхний уровень Actions, средний уровень – Left, а нижний – Right.
9. Щелкните на фрейме 20, выберите все три уровня и нажмите F5, чтобы добавить 20 фреймов для всех них. Щелкните на первом фрейме уровня Actions и на панели Actions введите stop().
10. Щелкните на первом фрейме уровня Left. Выберите средство рисования линий и на панели свойств установите толщину линии 6 пикселов, а цвет – зеленый. От базы в позиции X=0, Y=0 нарисуйте линию под углом –45 градусов длиной примерно 12 пикселов. Возможно, вам потребуется переместить базу в центральную позицию в редакторе символа.
11. Щелкните на первом фрейме уровня Right. Повторите шаг 10, но угол линии установите в +45 градусов. Когда вы закончите рисовать, вы должны увидеть V-образную картинку с основанием в центральной позиции (X=0, Y=0).
12. Выберите уровень Left, щелкните на фрейме 20 и нажмите F6, чтобы добавить ключевой фрейм. Выберите зеленую линию и измените ее цвет на пурпурный. Установите следующие размеры линии: ширина – 240, высота – 220; сместите основание линии в центральную позицию.

13. Повторите шаг 12 для линии на уровне Right, за исключением того, что размеры должны быть такими: ширина – 220, высота – 240. Вы должны увидеть большую пурпурную букву V, когда закончите рисовать.
14. Щелкните на первом фрейме для уровней Left и Right и выберите Shape for the Tween. Нажмите клавишу ввода, чтобы протестировать ваш клип. Вы должны увидеть «луч», палящий налево и направо и меняющий цвет с зеленого на пурпурный. Рисунок 8.5 показывает, как приблизительно должен выглядеть такой боевой луч, направленный на два космических корабля.

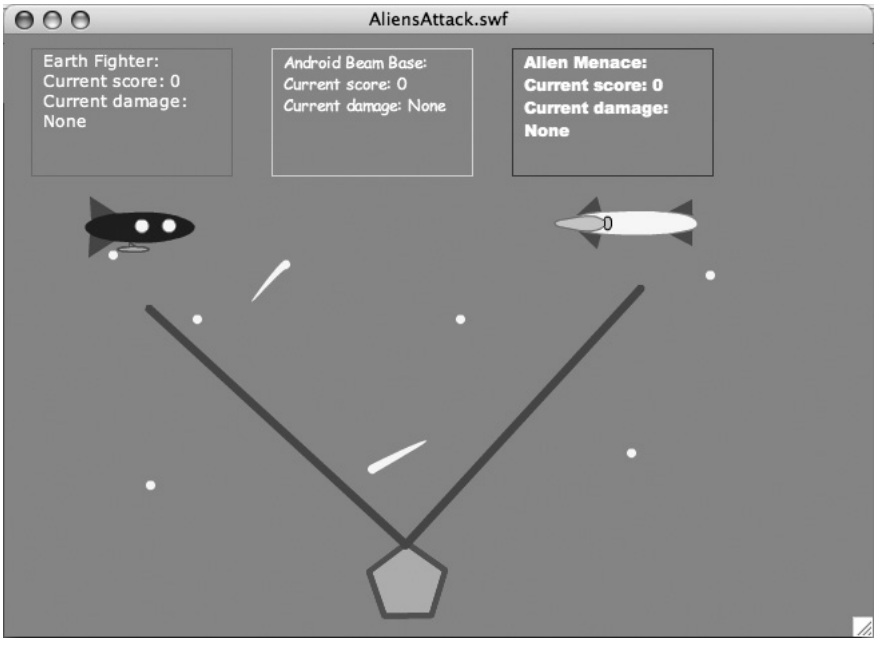


Рис. 8.5. *Андроид выстрелил боевым лучом по кораблям землянина и пришельца*

Теперь у вас должны быть готовы все классы и объекты, которые вам потребуются для боевой сцены. Проверьте, что на вашей панели Library имеются все следующие клипы:

- Alien
- Android
- Beam
- bullet
- Earthling
- Missile

- projectile
- Torpedo

Те объекты, чьи имена в данном списке начинаются с заглавной буквы, являются классами, а те, чьи имена начинаются с маленькой буквы, – объектами, принадлежащими одному из классов.

Написание обработчика изменений

При создании чего-то, что будет изменять условия и посылать сообщения землянину, пришельцу и андроиду, вам потребуется немного воображения. Если оружие врага или луч андроида попадает в пришельца или землянина, уровень поражения должен рассылаться всем подписчикам с помощью шаблона проектирования Наблюдатель. Каждое из окон, показывающих результаты, представляет различные языки, понятные для всех воюющих сторон. Когда один из сражающихся выбывает, он автоматически отписывается от рассылки и больше не получает новой информации, даже хотя он может испытывать последующие удары, отображаемые у выживших противников. (Станция андроида вообще неуязвима и может продолжать стрелять и получать информацию из рассылки, даже когда все остальные уже выбыли.)

Класс `ChangeHandler` разделен на несколько функциональных частей. Его конструктор создает объекты конкретной рассылки и конкретных наблюдателей. Каждому из наблюдателей дается собственное имя «участника войны» (`nomDeGuerre`), чтобы следить за всеми подписчиками, хранящимися в массиве `warrior` («воин»). Второй массив, `battleUpdate` («новости сражения»), хранит в себе боевые имена каждого из сражающихся для текстовых полей названий окошек их «информационных центров повреждений». Чтобы изобразить разные языки получаемых сообщений от конкретной рассылки, различные участники битвы имеют различные шрифт и цвет границы окошка информационного экрана. Функции `createTextWindow()` и `getFont()` занимаются обработкой отображаемых объектов.

Все объекты из панели Library размещаются на сцене вместе с информацией о выстрелах и повреждениях в довольно большой функции `getParts()` («собрать части»). Визуальные объекты согласованы с объектами конкретных наблюдателей.

Метод `setScore()` может вызываться объектами класса конкретной рассылки `ConcreteSubject`. При вызове метода `setScore()` все подписанные к рассылке наблюдатели получают уведомления через метод `passOn()` («передай»), который возвращает текущий счет через метод `update()`. Функция `killWarrior()` («убить воина») вызывает метод `unsubscribeObserver()`, так что, если чей-то космический корабль совсем разбит и выбыл из игры, он больше не получает новых сообщений.

Пример 8.19 содержит в себе довольно большой класс, и вы, возможно, захотите разбить его на более мелкие классы или упростить часть

функций. Тем не менее он призван продемонстрировать вам, как шаблон проектирования Наблюдатель может быть использован в ситуации, когда информация быстро меняется. Сохраните данный сценарий в файле *ChangeHandler.as* в том же каталоге, что и остальные файлы.

Пример 8.19. ChangeHandler.as

```
package
{
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.display.Sprite;

    public class ChangeHandler extends Sprite
    {
        private var eKiller:uint;
        private var aKiller:uint;
        private var aFlag:Number;
        private var aeFlag:Number;
        private var eFlag:Number;
        private var lFlag:Number;
        private var outputColor:Number;
        private var battleUpdate:Array=[];
        private var warrior:Array=[];
        private var scoreSetter:ConcreteSubject;
        private var earthling:ConcreteObserver;
        private var alien:ConcreteObserver;
        private var scoreFormat:TextFormat;
        private var missile:Missile;
        private var alienMC:Alien;
        private var torpedo:Torpedo;
        private var androidMC:Android;
        private var beam:Beam;
        private var earthlingMC:Earthling;

        // Главная функция
        public function ChangeHandler ()
        {
            scoreSetter=new ConcreteSubject();
            earthling=new ConcreteObserver(scoreSetter);
            warrior.push (earthling);
            earthling.nomDeGuerre="Earth Fighter";

            var android:ConcreteObserver=
                new ConcreteObserver(scoreSetter);
            warrior.push (android);
            android.nomDeGuerre="Android Beam Base";

            alien=new ConcreteObserver(scoreSetter);
```

```
        warrior.push (alien);
        alien.nomDeGuerre="Alien Menace";

        var eNm:String=earthling.nomDeGuerre.toLowerCase();
        var anNm:String=android.nomDeGuerre.toLowerCase();
        var alNm:String=alien.nomDeGuerre.toLowerCase();
        battleUpdate=new Array(3);
        scoreSetter.setScore (0,"None");
        createTextWindow ();
        getParts ();
    }
    // Создание окон вывода
    private function createTextWindow ():void
    {
        for (var i:int=0; i < battleUpdate.length; i++)
        {
            battleUpdate[i]=new TextField();
            addChild (battleUpdate[i]);
            battleUpdate[i].x=20* 1 + (i * 180);
            battleUpdate[i].y=10;
            battleUpdate[i].width=150;
            battleUpdate[i].height=95;
            battleUpdate[i].wordWrap=true;
            battleUpdate[i].multiline=true;
            battleUpdate[i].border=true;
            battleUpdate[i].textColor=0xffffee;
            scoreFormat=new TextFormat();
            scoreFormat.leftMargin=7;
            scoreFormat.rightMargin=7;
            getFont (warrior[i].nomDeGuerre);
            battleUpdate[i].borderColor=outputColor;
            battleUpdate[i].defaultTextFormat=scoreFormat;
            battleUpdate[i].text=warrior[i].nomDeGuerre+
                ":\n"+warrior[i].passOn();
        }
    }
    // Получение шрифта
    private function getFont (creature:String):void
    {
        switch (creature)
        {
            case "Earth Fighter" :
                outputColor=0xff0000;
                scoreFormat.font="Verdana";
                break;

            case "Android Beam Base" :
                outputColor=0x00ff00;
                scoreFormat.font="Comic Sans MS";
                break;

            case "Alien Menace" :
```

```
            outputColor=0x0000ff;
            scoreFormat.font="Arial Black";
            break;
    }
}
// Получение клипов из библиотеки
private function getParts ():void
{
    // Землянин
    earthlingMC=new Earthling();
    addChild (earthlingMC);
    earthlingMC.x=60,
    earthlingMC.y=120;
    missile=new Missile();
    addChild (missile);
    missile.x=earthlingMC.x+25;
    missile.y=earthlingMC.y+35;
    earthlingMC.addEventListener (MouseEvent.CLICK,fireMissile);
    earthlingMC.addEventListener
        (Event.ENTER_FRAME,earthlingHit);

    // Андроид
    androidMC=new Android();
    addChild (androidMC);
    androidMC.x=300;
    androidMC.y=380;
    beam=new Beam();
    addChild (beam);
    beam.x=androidMC.x;
    beam.y=androidMC.y;
    androidMC.addEventListener (MouseEvent.CLICK,fireBeam);
    androidMC.addEventListener (Event.ENTER_FRAME,beamHit);

    // Пришелец
    alienMC=new Alien();
    addChild (alienMC);
    torpedo=new Torpedo();
    addChild (torpedo);
    alienMC.x=420,
    alienMC.y=120;
    torpedo.x=alienMC.x-7;
    torpedo.y=alienMC.y+17;
    alienMC.addEventListener (MouseEvent.CLICK,fireTorpedo);
    alienMC.addEventListener (Event.ENTER_FRAME,alienHit);
}

// Землянин выпускает ракету
private function fireMissile (evt:Event)
{
    missile.play ();
    eFlag=0;
}
```

```
// Землянин попадает в пришельца
private function earthlingHit (evt:Event)
{
    this.alienMC=alienMC;
    if (missile.hitTestObject(alienMC) && eFlag==0)
    {
        eFlag=1;
        aKiller++;
        scoreSetter.setScore (aKiller,"Alien Hit");
        dataOut ();
        if (aKiller >= 5)
        {
            scoreSetter.setScore (aKiller,"Alien Out");
            alien.nomDeGuerre="Alien Destroyed";
            this.alienMC.rotation=90;
            torpedo.rotation=90;
            killWarrior (alien);
            dataOut ();
        }
    }
}
// Андроид стреляет лучом
private function fireBeam (evt:Event)
{
    beam.play ();
    aFlag=0;
    aeFlag=0;
}
// Андроид попадает в пришельца или землянина
private function beamHit (evt:Event)
{
    if (beam.hitTestObject(alienMC) && aFlag==0)
    {
        aFlag=1;
        aKiller++;
        scoreSetter.setScore (aKiller,"BeamHit on Alien");
        dataOut ();
        if (aKiller >= 5)
        {
            scoreSetter.setScore (aKiller,"Alien Out");
            alien.nomDeGuerre="Alien Destroyed";
            this.alienMC.rotation=90;
            torpedo.rotation=90;
            killWarrior (alien);
            dataOut ();
        }
    }
    if (beam.hitTestObject(earthlingMC) && aeFlag==0)
    {
        aeFlag=1;
        eKiller++;
```

```
            scoreSetter.setScore (eKiller,"BeamHit on Earthling");
            dataOut ();
            if (eKiller >= 5)
            {
                scoreSetter.setScore (eKiller,"Earthling Out");
                earthling.nomDeGuerre="Earthling Off";
                this.earthlingMC.rotation=90;
                missile.rotation=90;
                killWarrior (earthling);
                dataOut ();
            }
        }
    }
    // Пришелец выпускает торпеду
    private function fireTorpedo (evt:Event)
    {
        torpedo.play ();
        lFlag=0;
    }
    // Пришелец попадает в землянина
    private function alienHit (evt:Event)
    {
        this.earthlingMC=earthlingMC;
        if (torpedo.hitTestObject(earthlingMC) && lFlag==0)
        {
            lFlag=1;
            eKiller++;
            scoreSetter.setScore (eKiller,"Earthling Hit");
            dataOut ();
            if (eKiller >= 5)
            {
                scoreSetter.setScore (eKiller,"Earthling Out");
                earthling.nomDeGuerre="Earthling Off";
                this.earthlingMC.rotation=90;
                missile.rotation=90;
                killWarrior (earthling);
                dataOut ();
            }
        }
    }

    // Данные для вывода
    private function dataOut ()
    {
        for (var i:int = 0; i < battleUpdate.length; i++)
        {
            battleUpdate[i].text=warrior[i].nomDeGuerre+
                ":\n"+warrior[i].passOn();
        }
    }
    // Кладбище кораблей
```

```
            public function killWarrior (warship:ConcreteObserver)
            {
                scoreSetter.unsubscribeObserver (warship);
            }
        }
    }
```

Когда вы протестируете данный сценарий, вы обнаружите, что когда вы стреляете со станции андроида, видны только записи о попадании в землянина. Это происходит потому, что землянин и пришелец поражаются одним и тем же двойным боевым лучом, первым записывается и рассылается поражение пришельца, после чего тут же следует информация об ударе по землянину. В результате вы можете видеть только поражения, наносимые боевым лучом землянину. Рисунок 8.6 показывает момент игры, когда корабль пришельца разрушен, а корабль землянина все еще испытывает удары лучом андроида. Поскольку после разрушения пришелец перестает получать сообщения, он не показывает информацию о последующих ударах по землянину.

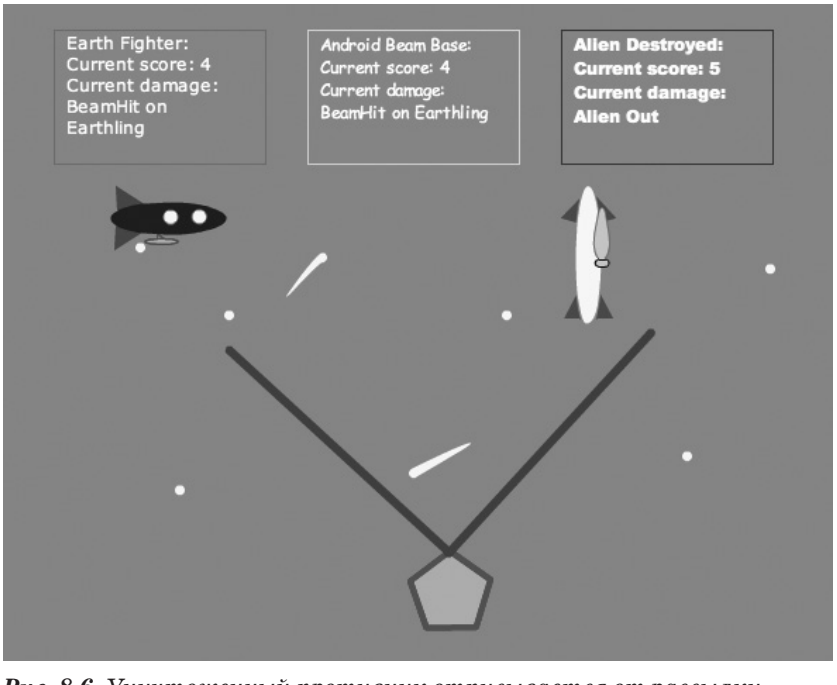


Рис. 8.6. *Уничтоженный противник отписывается от рассылки*

Хотя данный пример нельзя назвать реальной игрой, он наглядно демонстрирует отличную возможность рассылки информации о счете различным подписчикам в различных форматах. Кроме того, он демонстрирует, как шаблон проектирования Наблюдатель легко обрабатыва-

ет быстро меняющиеся данные одной рассылкой и быстро передает их всевозможным подписчикам.

Пример: различные отображения данных

Последний пример показал нам, как быстро меняющиеся данные могут быть посланы центральному объекту и переданы подписчикам. Кроме того, он продемонстрировал, как можно динамически отписывать подписчиков от рассылки. Мы также показали некоторые различия в отображении информации на экранах трех противников, но они были совсем незначительными. Теперь мы посмотрим на сильно различающиеся способы отображения информации и введем дополнительный интерфейс.

Дизайнер отображения информации

В своей первой книге о шаблонах проектирования «банда четырех» использовала простую схему для иллюстрации работы шаблона Наблюдатель. Один источник данных рассылал информацию различным объектам, отображающим ее в таблице, столбчатой диаграмме и круговой диаграмме. Такая схема оказалась хорошим образцом для демонстрации некоторых компонентов и графических возможностей Flash девятой версии и ActionScript 3.0, применяющих шаблон проектирования Наблюдатель. Для нашего примера давайте используем компонент отображения списка и покажем, как создавать столбчатую диаграмму и линейный график с помощью ActionScript.

Чтобы решение нашей задачи было более наглядным и могло повторно использоваться в других проектах, мы разобьем все способы отображения информации по разным классам. Пусть наши данные отображают квартальные показатели, тогда наше приложение будет принимать четыре числовых значения, по одному за каждый квартал. Кроме того, чтобы сделать процесс отображения информации более легко управляемым, мы введем третий интерфейс в дополнение к интерфейсам рассылки и наблюдателя. Этот интерфейс будет возвращать нам массив, состоящий из четырех числовых значений, по одному за каждый из кварталов. Сначала создайте три файла ActionScript и скопируйте в них примеры с 8.20 по 8.22. Сохраните эти файлы интерфейсов в одном каталоге, используя имена из заголовков примеров.

Пример 8.20. Subject.as

```
package
{
    // Интерфейс рассылки
    public interface Subject
    {
        function subscribeObserver(o:Observer):void;
        function unsubscribeObserver(o:Observer):void;
```

```
        function notifyObserver():void;
    }
}
```

Пример 8.21. Observer.as

```
package
{
    // Интерфейс наблюдателя
    public interface Observer
    {
        function update(q1:Number,q2:Number,q3:Number,q4:Number):
            void;
    }
}
```

Пример 8.22. DataOut.as

```
package
{
    // Интерфейс вывода данных
    public interface DataOut
    {
        function outToDesign():Array;
    }
}
```

Интерфейсы `Subject` и `Observer` вы уже видели ранее в данной главе. Упоминавшиеся нами четыре поквартальных числовых значения можно увидеть в интерфейсе наблюдателя `Observer`.

Конкретные классы и двойная имплементация

Классы конкретной рассылки и конкретного обозревателя немного изменились по сравнению с предыдущим примером. Мы не будем теперь использовать имена, поскольку объекты для отображения информации не похожи на людей. Так что наши классы будут немного проще, чем в предыдущих двух примерах. Теперь откройте два новых файла ActionScript и сохраните в них примеры 8.23 и 8.24, используя имена из их заголовков.

Пример 8.23. ConcreteSubject.as

```
package
{
    // Конкретная рассылка
    public class ConcreteSubject implements Subject
    {
        private var q1:Number,q2:Number,q3:Number,q4:Number;
        private var observers:Array;
        function ConcreteSubject():void
        {
            observers=new Array();
```

```
        }
        public function subscribeObserver(obserNow:Observer):void
        {
            observers.push(obserNow);
        }
        public function unsubscribeObserver(obserNow:Observer):void
        {
            for (var ob=0; ob<observers.length; ob++)
            {
                if (observers[ob]==obserNow)
                {
                    observers.splice(ob,1);
                }
            }
        }
        public function notifyObserver():void
        {
            for (var notify in observers)
            {
                observers[notify].update(q1,q2,q3,q4);
            }
        }
        public function setQuarter(q1:Number,q2:Number,
            q3:Number,q4:Number):void
        {
            this.q1=q1;
            this.q2=q2;
            this.q3=q3;
            this.q4=q4;
            notifyObserver();
        }
    }
}
```

Пример 8.24. ConcreteObserver.as

```
package
{
    // Конкретный наблюдатель
    class ConcreteObserver implements Observer,DataOut
    {
        private var dataNow:Array;
        private var q1:Number,q2:Number,q3:Number,q4:Number;
        public function ConcreteObserver()
        {
        }
        public function outToDesign():Array
        {
            return dataNow;
        }
        public function update(q1:Number,q2:Number,q3:Number,q4:
            Number):void
```

```
            {
                this.q1=q1;
                this.q2=q2;
                this.q3=q3;
                this.q4=q4;
                dataNow=new Array(q1,q2,q3,q4);
                outToDesign();
            }
        }
    }
```

Вы могли отметить, что в файле конкретного наблюдателя были имплементированы два интерфейса – `Observer` и `DataOut`. Первый из них имплементирует функцию `update()`, которая вызывает функцию `outToDesign()` из интерфейса `DataOut`. При этом вместо пустого типа `void` функция `outToDesign()` возвращает массив. Обратите внимание, что данные массива хранятся в переменной `dataNow` и возвращаются методом `outToDesign()`.

Классы отображения данных

Классы, используемые для трех различных видов отображения информации, потребуют более пристального нашего внимания. Не забывайте, что массив во всех трех классах является массивом со значениями за четыре квартала. Во всех трех классах массив называется одинаково, `listArray`, чтобы напомнить вам, что эти данные поступают из одной рассылки. Способы их отображения могут быть разными, но все данные, поступающие в каждый объект через подписку, совершенно идентичны.

Компонент UIList

Копия компонента `UIList` должна быть в вашей библиотеке `Library` при запуске файла документа Flash (*fla*-файла) для тестирования данного приложения. Прежде чем продолжать, откройте новый файл документа Flash и сохраните его как *DoDesign.fla* в том же каталоге, что и остальные файлы данного приложения. После этого выполните следующие шаги для создания этого компонента и подготовки его к использованию вместе с другими компонентами Flash 9 и ActionScript 3.0.

1. Откройте панели `Components` и `Library` из меню `Window` и перетащите компонент `List` с панели `Components` прямо на панель `Library`. После этого вы увидите, что в `Library` появились все сопутствующие файлы данного компонента.
2. Перетащите копию компонента `Button` на панель `Library`. Сейчас она вам не понадобится, это случится позднее.
3. На панели свойств `Properties` введите `DoDesign` в поле класс документа. Это будет наш главный класс приложения. Он вам потребуется, когда вы создадите все ваши классы.

4. Сохраните файл *DoDesign.fla*. Некоторое время он вам не потребуется, но держите его открытым, чтобы вы могли позже протестировать ваше приложение.

Это все, что вам нужно сделать с файлом документа Flash. Откройте теперь файл ActionScript и сохраните класс из примера 8.25, используя имя из его заголовка.

Пример 8.25. QuarterList.as

```
package
{
    // Установка компонента List для получения массива данных
    import flash.display.Sprite;
    import fl.controls.List;

    public class QuarterList extends Sprite
    {
        var listArray:Array;

        // Данные, отображаемые в компоненте List
        public function QuarterList(uData:Array)
        {
            listArray=new Array();
            listArray=uData;
            var quarter_list:List=new List();
            quarter_list.addItem({label:"Quarterly Results"});
            for (var quarter in listArray)
            {
                quarter_list.addItem({label:"Q"+(quarter+1)+"=
                    "+listArray[quarter],data:listArray[quarter]});
            }
            quarter_list.rowCount=5;
            quarter_list.width=110;
            addChild(quarter_list);
            quarter_list.x=50;
            quarter_list.y=55;
        }
    }
}
```

В данном приложении вам требуются только строковые названия, которые будут показывать квартальные данные. Однако в других приложениях вам могут понадобиться и числовые данные, поэтому данный класс хранит в себе и строковые, и числовые данные. Одно строковое название нужно для заголовка списка, так что первое строковое значение, «*Quarterly Results*» («квартальные результаты»), не изменяется и служит общим названием для всех данных компонента `UIList`. Рису-

нок 8.7 показывает нам пример того, как может выглядеть отображение данных в нашем списке.

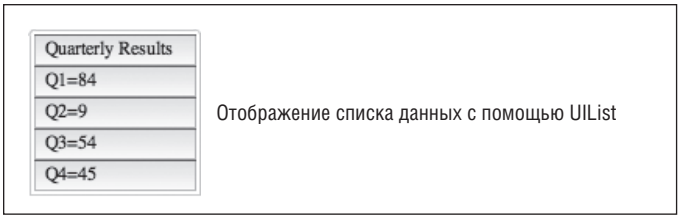


Рис. 8.7. *Данные в списке*

Как нами уже отмечалось, хотя вы не можете увидеть исходных данных для отображения, они тоже хранятся в нашем компоненте. Для некоторых приложений они могут потребоваться.

Столбчатая диаграмма

Столбчатая диаграмма имеет свойство, которого нет у списка данных: она показывает относительную величину данных. Экран компьютера имеет ограниченное число пикселов для показа столбчатых диаграмм. Если ваши значения для нее относительно малы, скажем, между 1 и 100, то у вас достаточно места для их непосредственного показа. Но когда ваши числа становятся больше, у вас для них просто не хватает вертикальных и горизонтальных размеров экрана. Так как данные по кварталам должны отображаться как относительные по отношению друг к другу величины, наша задача становится немного проще. Нам только надо найти самое большое из четырех чисел в нашем массиве. Далее мы будем рассматривать его как множитель, представляющий наибольшее значение в нашей диаграмме. Например, предположим, что мы создаем столбчатую диаграмму высотой не более 200 пикселов, и у нас есть следующие числа: 320, 432, 121, 89. Максимальное число здесь 432. Оно должно быть представлено столбцом высотой не более 200 пикселов. Используя формулу:

```
(currentValue /; MaxValue) x maxPixels
```

мы можем найти высоту каждого столбца. Как вы понимаете, максимальной высотой будет 200 пикселов, а все столбцы для меньших значений в нашей группе чисел будут пропорционально ниже:

```
432/; 432 = 1 x 200 = 200
121/; 432 =.28 x 200 =56
```

После этого нам остается лишь нарисовать прямоугольники, представляющие полученные значения. Откройте новый файл ActionScript и сохраните пример 8.26, используя имя файла, указанное в его заголовке, в той же самой папке, что и остальные файлы данного приложения.

Пример 8.26. QuarterBar.as

```
package
{
    // Создатель столбчатой диаграммы
    import flash.display.Graphics;
    import flash.display.Shape;
    import flash.display.Sprite;

    public class QuarterBar extends Sprite
    {
        private var listArray:Array;
        private var maxVal:Number=0;
        private var b1:Number, b2:Number, b3:Number,b4:Number;
        private var position:uint=160;
        private var maxSize:uint=150;

        // Информация, отображаемая в диаграмме
        function QuarterBar(cData:Array)
        {
            listArray=new Array();
            listArray=cData;

            // Установка относительных размеров столбцов
            for (var max in listArray)
            {
                if (listArray[max] > maxVal)
                {
                    maxVal=listArray[max];
                }
            }
            b1=(listArray[0]/maxVal)*maxSize;
            b2=(listArray[1]/maxVal)*maxSize;
            b3=(listArray[2]/maxVal)*maxSize;
            b4=(listArray[3]/maxVal)*maxSize;

            // Диаграмма
            var bar:Shape=new Shape();
            bar.graphics.clear();
            // Рисуем
            bar.graphics.beginFill(0xdd0000);
            bar.graphics.lineStyle(1,0x000000);
            bar.graphics.drawRect(285,(position-b1),30,b1);
            bar.graphics.drawRect(315,(position-b2),30,b2);
            bar.graphics.drawRect(345,(position-b3),30,b3);
            bar.graphics.drawRect(375,(position-b4),30,b4);
            bar.graphics.endFill();

            addChild(bar);
        }
    }
}
```

Рисунок 8.8 показывает, как будет выглядеть столбчатая диаграмма с нашими значениями.

***Рис. 8.8.** Столбчатая диаграмма*

Будучи один раз нарисованным, графическое изображение диаграммы не очистится само со сцены при замене его новым изображением. Метод `clear()` просто очищает текущую память, но не сцену. Для очистки сцены нужно использовать `removeChild()` («удалить дочерний узел»), что требует указания на добавленный элемент. Поэтому в главной программе мы использовали специальный признак того, что графический элемент был добавлен; при новом наборе данных этот графический элемент удаляется и признак очищается.

Линейный график

Для линейного графика нужны те же пропорциональные точки для рисования линий, что и для рисования прямоугольников в столбчатой диаграмме. К счастью, они вычисляются тем же способом. Дополнительной особенностью линейного графика является наличие на нем некоторой фоновой координатной сетки, которая позволяет лучше оценивать его линии. Линии графика без такой сетки выглядят разрозненными, свободно парящими в пространстве. Поэтому в дополнение к самому линейному графику данный класс создает для него соответствующую координатную сетку. Следуя рекомендациям дизайнера представления информации Эдварда Тафта (Edward Tufte), что координатная сетка должна минимально отличаться от фона, мы используем для нее тонкие линии светло-серого цвета. А для самого графика мы будем использовать более толстые линии синего цвета. Таким образом, координатная сетка не будет мешать его восприятию, но будет обеспечивать для него правильный контекст. Откройте новый файл ActionScript, введите в него код из примера 8.27 и сохраните его под именем из заголовка этого примера.

Пример 8.27. QuarterGraph.as

```
package
{
    // Создатель линейного графика
```

```
import flash.display.Graphics;
import flash.display.Shape;
import flash.display.Sprite;

public class QuarterGraph extends Sprite
{
    private var listArray:Array;
    private var maxVal:Number=0;
    private var ln1:Number, ln2:Number, ln3:Number,ln4:Number;
    private var position:uint=300;
    private var maxSize:uint=100;

    // Устанавливаем относительные размеры линий
    function QuarterGraph (lData:Array)
    {
        listArray=new Array();
        listArray=lData;

        // Устанавливаем размеры
        for (var max in listArray)
        {
            if (listArray[max] > maxVal)
            {
                maxVal=listArray[max];
            }
        }
        ln1=(listArray[0]/maxVal)*maxSize;
        ln2=(listArray[1]/maxVal)*maxSize;
        ln3=(listArray[2]/maxVal)*maxSize;
        ln4=(listArray[3]/maxVal)*maxSize;

        // Данные, отображаемые в графике
        var line:Shape=new Shape();
        // Рисуем контур
        line.graphics.lineStyle (.25,0xcccccc);
        line.graphics.moveTo (50,200);
        line.graphics.lineTo (50,position);
        line.graphics.lineTo (200,position);
        line.graphics.moveTo (100,200);
        line.graphics.lineTo (100,position);
        line.graphics.moveTo (150,200);
        line.graphics.lineTo (150,position);
        line.graphics.moveTo (200,200);
        line.graphics.lineTo (200,position);
        line.graphics.moveTo (50,200);
        line.graphics.lineTo (200,200);
        line.graphics.moveTo (50,250);
        line.graphics.lineTo (200,250);
        line.graphics.moveTo (50,275);
        line.graphics.lineTo (200,275);
        line.graphics.moveTo (50,225);
```

```
                line.graphics.lineTo (200,225);

                // Рисуем график
                line.graphics.lineStyle (2,0x0000cc);
                line.graphics.moveTo (50,(position-ln1));
                line.graphics.lineTo (100,(position-ln2));
                line.graphics.lineTo (150,(position-ln3));
                line.graphics.lineTo (200,(position-ln4));

                addChild (line);
            }
        }
    }
```

Рисунок 8.9 показывает наш линейный график на фоне координатной сетки.

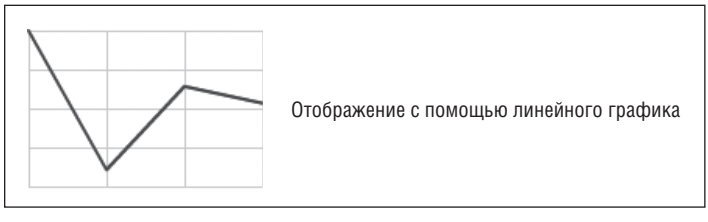


Рис. 8.9. *Линейный график*

Если вам когда-либо потребуется наложить графики друг на друга для демонстрации изменений или для сравнения, то просто *не* используйте в главной программе функцию `removeChild()` для перерисовки вашего графика.

Сборка всех элементов вместе

При наличии 10 различных файлов задача сборки всех их в один класс может показаться несколько устрашающей. Однако вы убедитесь, что это не так сложно. Классы сами выполняют большую часть всей тяжелой работы, и вам остается только воспользоваться ими.

Самой важной особенностью класса `DataDesign` является ввод данных в форме текстовых полей. Как и в примере с космической битвой, различные текстовые поля помещаются в один массив элементов. Однако они потом используются не для отображения данных, а для передачи их значений. Значения этих текстовых полей конвертируются в числа и присваиваются числовым переменным. Числовые переменные передаются объекту конкретной рассылки, и метод-установщик делает из них текущие ее значения. Далее эти значения рассылаются конкретным наблюдателям в форме массива и используются для отображения данных. Откройте новый файл ActionScript, введите в него код из примера 8.28 и сохраните его под именем из заголовка этого примера.

Пример 8.28. DataDesign.as

```
package
{
    // Главная программа
    import flash.display.Sprite;
    import fl.controls.Button;
    // Текстовые поля для меток
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    import flash.text.TextFieldType;
    // События
    import flash.events.MouseEvent;
    import flash.events.Event;

    public class DataDesign extends Sprite
    {
        private var dataEntry:Array=[];
        private var quarterGraph:QuarterGraph;
        private var quarterBar:QuarterBar;
        private var quarterList:QuarterList;
        private var dataSub:ConcreteSubject;
        private var listDisplay:ConcreteObserver;
        private var barChart:ConcreteObserver;
        private var lineGraph:ConcreteObserver;
        private var xpos:uint=250;
        private var ypos:uint=250;
        private var dt1:Number,dt2:Number,dt3:Number,dt4:Number;
        private var dataBtn:Button;
        private var barFlag:Boolean=true;

        public function DataDesign ()
        {
            // Создаем конкретные рассылки и наблюдателей
            dataSub=new ConcreteSubject();
            listDisplay=new ConcreteObserver();
            barChart=new ConcreteObserver();
            lineGraph=new ConcreteObserver();
            dataSub.subscribeObserver (listDisplay);
            dataSub.subscribeObserver (barChart);
            dataSub.subscribeObserver (lineGraph);

            doText ();
            doDataEntry ();
            doButton ();
            dataBtn.addEventListener (MouseEvent.CLICK,showData);

        }
        // Вывод данных на сцену
        function showData (ev:Event):void
        {
            if (barFlag)
```

```
        {
            barFlag=false;
        }
        else
        {
            removeChild (quarterBar);
            removeChild (quarterGraph);
        }
        //doObservers();
        dt1=Number(dataEntry[0].text);
        dt2=Number(dataEntry[1].text);
        dt3=Number(dataEntry[2].text);
        dt4=Number(dataEntry[3].text);
        doDisplay (dt1,dt2,dt3,dt4);

        // Данные, отображаемые в компоненте List
        quarterList=new QuarterList(listDisplay.outToDesign());
        addChild (quarterList);

        // Данные, отображаемые в столбчатой диаграмме
        quarterBar=new QuarterBar(barChart.outToDesign());
        addChild (quarterBar);

        // Данные, отображаемые в линейном графике
        quarterGraph=new QuarterGraph(lineGraph.outToDesign());
        addChild (quarterGraph);
    }
    // Добавляем метки
    private function doText ():void
    {
        var ui:TextField=new TextField();
        var bar:TextField=new TextField();
        var ln:TextField=new TextField();
        var dt:TextField=new TextField();
        ui.autoSize=TextFieldAutoSize.LEFT;
        bar.autoSize=TextFieldAutoSize.LEFT;
        ln.autoSize=TextFieldAutoSize.LEFT;
        addChild(ui),addChild(bar),addChild(ln);
        addChild (dt);
        ui.x=50, bar.x=285,ln.x=50;
        dt.x=xpos;
        ui.y=165, bar.y=165,ln.y=305;
        dt.y=ypos+15;
        ui.text="List UI Display";
        bar.text="Bar Chart Display";
        ln.text="Line Graph Display";
        dt.text="Enter Data";
    }
    // Функция ввода данных
    private function doDisplay (n1:Number,n2:Number,
        n3:Number,n4:Number):void
```

```
            {
                dataSub.setQuarter (n1,n2,n3,n4);
            }
            // Ввод данных
            private function doDataEntry ():void
            {
                for (var de=0; de<4; de++)
                {
                    dataEntry[de]=new TextField();
                    dataEntry[de].border=true;
                    dataEntry[de].borderColor=0x999999;
                    dataEntry[de].background=true;
                    dataEntry[de].backgroundColor=0xcccccc;
                    dataEntry[de].type=TextFieldType.INPUT;
                    dataEntry[de].width=32;
                    dataEntry[de].height=18;
                    this.addChild (dataEntry[de]);
                    dataEntry[de].x=xpos+(40*de);
                    dataEntry[de].y=ypos;
                }
            }
            // Компонент кнопки
            private function doButton ():void
            {
                dataBtn=new Button();
                dataBtn.label="Show Data";
                this.addChild (dataBtn);
                dataBtn.x=xpos;
                dataBtn.y=ypos+30;
            }
        }
    }
```

При обсуждении классов столбчатой диаграммы и линейного графика мы уже упоминали, что если вы не очистите существующее изображение с помощью функции `removeChild()`, то оно так и останется на сцене. С помощью признака `barFlag` класс проверяет, что изображение диаграммы или графика было добавлено на сцену, перед тем как пытаться удалить его. Без использования подобного признака у нас возникла бы ошибка при попытке удалить объект, который до этого не был добавлен с помощью `addChild()`.

Вы можете создать еще несколько классов для отображения данных и другими способами. Однако, чтобы ясно наблюдать процесс, как все данные вводятся и передаются конкретной рассылке для передачи всем различным объектам отображения, мы ограничили рост нашего приложения.

Заключение

Главной проверкой всех шаблонов проектирования является их гибкость. Как вы видели в примерах данной главы, во всех их классах рассылки и наблюдателя было очень мало изменений. Поскольку главной задачей данного шаблона проектирования является передача информации об изменении состояния нескольким различным наблюдателям, он применим во многих типах приложений. Благодаря его относительной простоте и гибкости, шаблон Наблюдатель очень широко используется при создании программ.

Очень важно также, что шаблон Наблюдатель позволил нам продемонстрировать, как важно использовать при программировании интерфейсы, а не их имплементации. В примерах 8.1 и 8.11 этот процесс был показан очень наглядно. В них каждый конкретный наблюдатель объявлялся с помощью его супертипа (интерфейса) `Observer`. Однако в некоторых примерах реализации шаблона Наблюдатель мы не могли использовать в имплементации супертип, как в примере 8.19. Это конечно не означает, что данный пример неправильный. Просто в некоторых случаях применения шаблона Наблюдатель возникают определенные проблемы при попытке использовать интерфейс, связанные с характеристиками данного интерфейса и его имплементации. В результате в подобных случаях приходится создавать объекты, применяя тип имплементации, а не интерфейса.

Раздумывая о том, где вы можете применить шаблон проектирования Наблюдатель, вспомните все ситуации, где информация о ключевых состояниях в программе должна быть разослана различным сущностям. Одним из самых распространенных примеров такой имплементации является случай, когда одни и те же данные должны быть переданы объектам для различных способов ее отображения на экране или интерпретации. Использование шаблона Наблюдатель гарантирует, что одинаковая информация будет разослана всем заинтересованным сторонам без создания множества отдельных источников информации.

9

Шаблонный метод

Когда вы пишете сообщение в свой блог,
вы создаете исторический документ.
Если вы меняете шаблон оформления,
ваше сообщение принимает совершенно
иной вид. Слова те же, но смысл – другой.
Все зависит от тех вопросов, которые
будут задавать люди, читая вашу запись,
и мы не можем знать, чего они захотят.

Джош Гринберг

Ну, я проделал всю подготовительную работу
и записал демонстрашки к песням, потом притащил их
в студию и проиграл парням. А затем мы взяли все это
за шаблон и быстро сбацали новый альбом.

Дэвид Санборн

Что такое Шаблонный метод?

Вам понравится этот шаблон проектирования. По существу *Шаблонный метод* – это состоящий из последовательности операций алгоритм, предназначенный для достижения определенной цели. Представьте себе некоторый набор действий, который надо выполнить, чтобы что-то сделать, например, приехать на работу. Если вы будете использовать для этого машину, яхту или самолет, то некоторые ваши действия будут совпадать для всех данных случаев, а некоторые – отличаться. Вы дойдете до вашего транспортного средства, направите его к месту назначения и выпьете, как обычно, взятую с собой чашку кофе. Таким об-

разом, если составлять общий шаблон ваших действий, охватывающий и машину, и яхту, и самолет, то ваш план (алгоритм) будет выглядеть приблизительно следующим образом:

1. Подготовить транспортное средство.
2. Направить транспортное средство.
3. Выпить чашку кофе.

Подготовка автомобиля к поездке обычно включает в себя как минимум проверку, что в баке достаточно бензина и шины у него не спущены. В случае использования яхты вы, возможно, захотите убедиться, что имеются все необходимые карты, и все средства спасения в хорошем состоянии. Полет на самолете требует его предварительного осмотра с листом проверки в руках. Итак, для выполнения первого шага вашего плана вам необходимы отдельные инструкции, зависящие от типа вашего транспортного средства.

В этой главе используются два вида названия «шаблонный метод». Название такого шаблона проектирования будет писаться с большой буквы: Шаблонный метод. А название функции шаблонный метод – с маленькой буквы.

Для выполнения второго шага плана поездки требуется вывести автомобиль на дорогу, сделать все необходимые повороты и проехать по нужным улицам. Временами это может быть весьма не просто, и водитель, вероятно, пожелает прослушать сообщения о ситуации на дорогах, чтобы избежать мест скопления транспорта или ремонта трасс. В зависимости от конкретных обстоятельств поездка на яхте может быть и простой поездкой по озеру, и сложным прохождением среди других судов, мелей и течений. Наконец, путешествие на самолете может быть как простым полетом по прямой в ясный день, так и сложным полетом по приборам, требующим знания различных средств навигации, в пасмурную погоду. Для выполнения второго шага плана, как и первого, необходимы специальные инструкции.

Для последнего шага вашего плана у вас есть чашка кофе, которую надо выпить по дороге на работу. Она одинакова для всех транспортных средств, какое бы из них вы ни выбрали. Вы используете одну и ту же термокружку для всех видов поездок, и нужно лишь постараться не пролить кофе себе на колени. В отличие от первых двух шагов плана, которые требовали специфических инструкций для каждого транспортного средства, данный шаг будет одинаковым для всех трех средств передвижения.

Создание шаблона действий для этого набора обстоятельств требует приблизительного описания методики для первых двух шагов и конкретной методики для последнего шага. При использовании для его проектирования Шаблонного метода все три вышеописанные функции будут помещены в один метод так, что их последовательность будет со-

блюдена в любом случае. Такой метод, содержащий упорядоченную последовательность функций, называется *шаблонным методом*, – методом, сохраняющим указанную последовательность действий. Вот как он приблизительно будет выглядеть в псевдокоде:

```
// Внутри абстрактного класса

// Это Шаблонный метод
final function templateMethod()
{
    prepareTransportation();
    navigatVehicle();
    drinkCoffee();
}

// Абстрактные функции
function prepareTransportation():void
{
    // пусто
}

function navigateVehicle(): void
{
    // пусто
}
// Конкретная функция
function drinkCoffee()
{
    trace("Drinking coffee on the way to work ");
}
// Конец класса
```

Обратите внимание, что Шаблонный метод не требует наличия всех конкретных функций. Однако приведенный пример иллюстрирует, что если таковые существуют для всех объектов данного класса, то весьма полезно их сразу включить в данный шаблон.

Ключевые характеристики

В общем Шаблонный метод – это гибкий создатель алгоритмов, строящихся на основе базового класса, где и определен указанный алгоритм. Данный шаблон проектирования имеет следующие основные характеристики:

- Использует наследование свойств для распространения алгоритма между классами
- Позволяет подклассам уточнять некоторые детали общего алгоритма
- Использует инвертированную структуру контроля – родительский класс вызывает операции в подклассе

- Определяет состав шагов алгоритма и фиксирует их последовательность в алгоритме
- Позволяет иметь опциональные подключаемые операции для расширений в фиксированных местах алгоритма

В группе шаблонов поведения «банда четырех» указала только два шаблона, основанных на *наследовании*, а не на *композиции*, и Шаблонный метод один из них. При его применении важно помнить, что, поскольку ActionScript 3.0 напрямую не поддерживает абстрактные классы, имплементировать главный абстрактный класс данного шаблона нельзя. Однако вы можете использовать при объявлениях в программе интерфейсы абстрактных классов (супертипы), а не их имплементации. Примеры в данной главе продемонстрируют сказанное.

Позволяя подклассам наполнять действия общего алгоритма новым содержанием, данный шаблон проектирования учитывает преимущества наследования и гибкость композиции. По существу Шаблонный метод использует наследование для изменения частей своего алгоритма. Поскольку его операции требуют детального наполнения в подклассах, мы можем рассматривать их как *зарезервированные места*, ждущие своего содержимого от подклассов. Выборочно замещая операции алгоритма (т. е. шаблонного метода), каждый подкласс может получить от него ту функциональность, которая ему необходима.

Инвертированную структуру контроля иногда называют *Принципом Голливуда*, поскольку родительский класс как бы объявляет своим дочерним классам (подклассам): «Не звоните нам, мы сами вас вызовем». Это превращает общий алгоритм данного абстрактного класса в инициатора всех действий. Родительский класс вызывает операции в подклассах в своем шаблонном методе. Дочерние классы просто обеспечивают все детали операций из алгоритма, контролируемого родительским классом.

Благодаря атрибуту `final` шаблонный метод является *фиксированным* методом. Функции с таким атрибутом не могут быть замещены в подклассах. На первый взгляд, это противоречит всему, что было нами сказано о том, что детали операций прописываются в подклассах путем их замещения там. Однако фиксируется только общий шаблонный метод; абстрактные функции операций не фиксируются в родительском классе. Рисунок 9.1 иллюстрирует такое построение.

Как вы видите на этом рисунке, операции, составляющие шаблонный метод, не фиксированы. Поэтому подклассы могут замещать операции, как им необходимо. Однако подклассы не могут изменить порядок следования операций, составляющих шаблонный метод.

Наконец, вы можете добавлять опциональные *подключаемые* операции. С одной стороны, такая операция – это не более чем еще один метод, который может быть замещен в общем контексте Шаблонного метода. С другой стороны, ее можно рассматривать как служебный про-

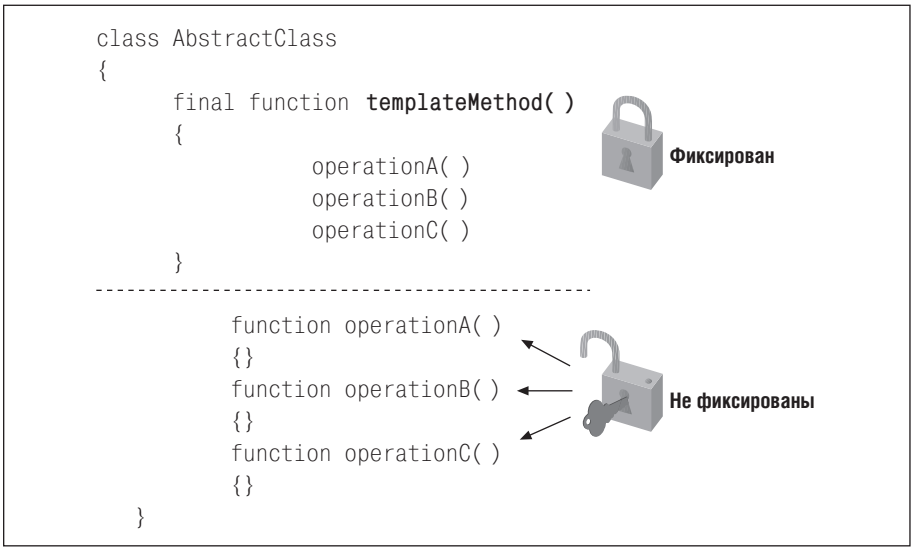


***Рис. 9.1.** Фиксированный шаблонный метод*

ход, через который имплементация (подкласс) может вмешаться в ход выполнения общего алгоритма. Поскольку концепцию подключаемых операций лучше всего изучать при рассмотрении примера, их обсуждение будет продолжено позднее в данной главе.

Модель шаблонного метода

При первом знакомстве с Шаблонным методом многие разработчики бывают поражены простотой и ясностью этого шаблона проектирования, а позднее и его практичностью. Хотя его схема, изображенная на диаграмме классов на рис. 9.2, очень проста, он может применяться для решения широкого круга задач.

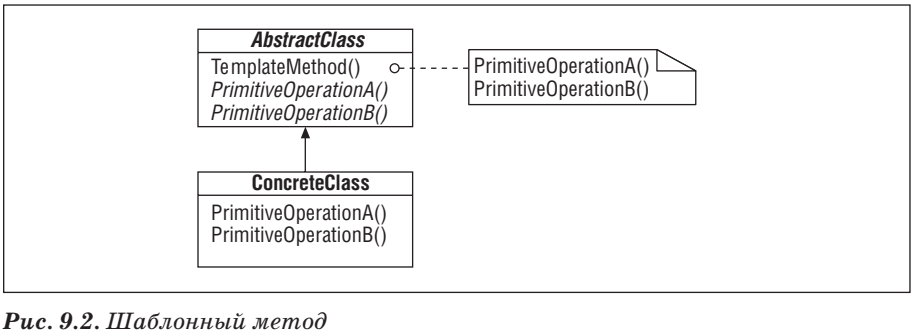


***Рис. 9.2.** Шаблонный метод*

Шаблонный метод очень просто понять и освоить в применении. Однако, как вы увидите, у него есть несколько особенностей, которые де-

лают его уникальным. Для использования Шаблонного метода важно разобраться в разнице между абстрактными функциями (операциями), которые должны быть замещены; конкретными функциями, неизменно составляющими общий алгоритм, которые не могут быть замещены; и подключаемыми операциями, которые могут быть опционально замещены. Проработав все примеры, вы сможете лучше уяснить ее.

Ключевые концепции ООП, используемые в Шаблонном методе

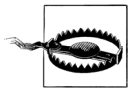
Похоже, нам будет совсем не просто назвать Шаблонный метод образцом хорошего применения ООП, поскольку он является одним из двух поведенческих шаблонов, где используется наследование, а не композиция. Ведь второй принцип, установленный «бандой четырех», говорит о *предпочтении композиции наследованию*. Почему же мы вообще рассматриваем шаблон, в котором все происходит наоборот? Кроме того, возникает вопрос по поводу *Голливудского принципа*: он уникален только для шаблонов, использующих наследование, или применим и в шаблонах, основанных на композиции?

Почему наследование, а не композиция?

Когда бы в шаблоне проектирования ни использовалось наследование вместо композиции, на это должна быть веская причина. Чтобы разобраться в ней, нужно полностью понимать сам принцип предпочтения композиции наследованию как общую рекомендацию по применению ООП. Этот принцип был установлен ввиду некоторых преимуществ композиции перед наследованием, особенно того из них, что композиция не нарушает инкапсуляцию. Однако Гамма (Erich Gamma), Хелм (Richard Helm), Джонсон (Ralph Johnson) и Влиссидес (John Vlissides) («банда четырех») отмечали, что наследование также имеет ряд положительных свойств. Одним из преимуществ наследования является то, что при использовании подклассов, где *некоторые, но не все* операции замещены, модификация такой многократно используемой имплементации гораздо проще. Поскольку Шаблонный метод делает именно это – замещает некоторые, но не все операции для достижения гибкости, – он использует вышеназванное ключевое преимущество наследования перед композицией.

Далее, обсуждая преимущества композиции перед наследованием, «банда четырех» отмечала, что процесс создания новых компонентов приложения путем повторного использования кода с помощью наследования легче, чем путем применения композиции существующих. Поэтому рекомендация по использованию композиции вместо наследования должна ограничиваться пониманием того, что на самом деле композиция и наследование работают вместе. Это не означает, что данный принцип

неправильный, а говорит лишь о том, что решение о его применении должно приниматься в контексте конкретной ситуации разработки.



При изучении всех шаблонов проектирования и принципов программирования, которые их окружают, не нужно заучивать эти принципы, словно мантры. Повторение принципов снова и снова может помочь вам запомнить их, но не обязательно приведет вас к их пониманию. Все они применимы только в определенных контекстах, которые очень важно понимать для их эффективного использования.

Абстрактные функции и их замещение

В предыдущих главах мы уже немного поворчали на тему, что в ActionScript 3.0 не существует абстрактных классов. Но в нем нет еще и *абстрактных функций*. Вместо того чтобы негодовать по этому поводу, давайте обсудим более важный вопрос, зачем и как их нужно использовать. Абстрактная функция – это способ редуцирования обычной функции до ее концепции или общей идеи без конкретного содержания. На практике сказанное означает создание имени функции и размещение ее в шаблонном методе, или главном вашем алгоритме, в нужном порядке, а не как объявление функции в интерфейсе. В языках типа Java вы можете написать:

```
abstract myAbstractFunction();
```

и все. В ActionScript 3.0 вы можете выполнить то же самое с помощью:

```
function myAbstractFunction() {}
```

Таким образом, хотя у нас и нет функций, которые можно было бы явно обозначить как абстрактные, вы можете эффективно создавать все нужные вам абстрактные функции, так же как и абстрактные классы, в ActionScript 3.0.

Кроме того факта, что в ActionScript 3.0 можно создавать функции, которые ведут себя как абстрактные, очень важно понять принцип их использования в контексте Шаблонного метода. Главными идеями создания всех шаблонов проектирования являются повторное использование кода и гибкость создаваемых приложений. Абстрактные функции *обязательно должны* замещаться, и это определяет их изменчивость и гибкость; при каждом замещении она неизбежно меняется. Если в родительском классе задействованы абстрактные функции, его подклассы могут использовать их для реализации своей необходимой имплементации алгоритма шаблонного метода. Повторному использованию кода благоприятствует то обстоятельство, что при использовании шаблона алгоритма у разработчика есть много средств по его изменению, поскольку абстрактные функции, составляющие шаблонный метод, могут быть модифицированы под требования конкретного приложения.

Учет вариаций

При выборе шаблона проектирования для приложения разработчикам следует *продумывать то, что должно измениться* в структуре этого приложения. Здесь речь идет не об изменениях, связанных с модернизацией приложения в будущем, а о том, что вы сами хотите изменить. В изучаемом шаблоне проектирования *шаблонный метод инкапсулирует в себе все части, составляющие его основной алгоритм*. Каждая абстрактная операция в шаблонном методе изменяется только в общем контексте этого алгоритма. Поскольку данный метод фиксирован, последовательность операций не изменяется, чего нельзя сказать о них самих. Таким образом, в Шаблонном методе отлично демонстрируется общий принцип шаблонов проектирования: *инкапсулировать изменяющееся*.

Принцип Голливуда

Директор по кастингу на фильм всегда говорит проходящим отбор актерам: «Мы дадим вам знать. Не звоните нам, мы сами вам позвоним». Шаблонный метод поступает точно так же: он оставляет свой метод в родительском классе, который вызывает свои подклассы. Если говорить более точно, шаблонный метод может вызвать: 1) конкретные операции, 2) конкретные операции из главного абстрактного класса (где определен сам шаблонный метод), 3) примитивы операций, 4) фабричный метод (см. главу 2) и 5) подключаемые функции. Однако все эти объекты более низкого уровня не могут вызывать родительские классы.

Поскольку «банда четырех» упоминала Голливудский принцип только при обсуждении Шаблонного метода, мы задались вопросом, применим ли данный принцип ко всем шаблонам проектирования или только к некоторым, с инвертированной структурой контроля. Если вы вернетесь назад к главе 2 и посмотрите на шаблон Фабричный метод или к главе 8 с шаблоном Наблюдатель, вы там тоже увидите работу данного принципа. Например, инвертированная структура контроля в шаблоне Наблюдатель информирует всех своих подписчиков об изменениях состояния. Информация при этом передается сверху вниз, к объектам более низкого уровня.

Так какова же задача этого принципа? Какое отношение он имеет к повторному использованию кода? Эрик и Элизабет Фриман (Eric Freeman, Elisabeth Freeman) элегантно вложили ответы на эти вопросы в фразу: способ предотвратить *деградацию зависимостей*. Другими словами, данный шаблон проектирования ослабляет зависимости компонентов более высокого уровня от низкоуровневых компонентов и всех связанных с ним зависимостей. Все это делает работу вашей системы более понятной. Рисунок 9.3 иллюстрирует обсуждаемый принцип.

Рис. 9.3. *Голливудский принцип*

Минималистский пример: абстрактный Шаблонный метод

Хотя и не имеет особого смысла реализовывать Шаблонный метод только с одним подклассом, такая конструкция позволит нам наглядно продемонстрировать его структуру. Этому и будет посвящен наш минималистский пример Шаблонного метода. Для него нам потребуется совсем немного: только два основных класса:

- Абстрактный класс с алгоритмом в шаблонном методе
- По крайней мере один конкретный класс как подкласс абстрактного класса

Обратите внимание на операции, которые будут уточняться в подклассах, и на одну операцию, которая будет постоянной для всех подклассов.

Костяк Шаблонного метода

Примеры 9.1 и 9.2 демонстрируют нам два класса, которые нужны для начала работы с этим простым, но полезным шаблоном. Сохраните каждый из примеров, используя имена из их заголовков.

Пример 9.1. AbstractClass.as

```
package
{
    // Абстрактный класс
    class AbstractClass
```

```
    {
        public final function templateMethod():void
        {
            primitiveA();
            primitiveB();
            fromTemplate();
        }
        protected function primitiveA():void
        {
            // для замещения
        }
        protected function primitiveB():void
        {
            // для замещения
        }
        private final function fromTemplate():void
        {
            trace("Hello everybody!");
        }
    };
}
```

Пример 9.2. ConcreteClass.as

```
package
{
    // Любой конкретный класс
    public class ConcreteClass extends AbstractClass
    {
        trace("Concrete class");
        override protected function primitiveA():void
        {
            trace("Special A");
        }
        override protected function primitiveB():void
        {
            trace("Special B");
        }
    }
}
```

Посмотрев на наш абстрактный класс, вы можете увидеть, что основной алгоритм инкапсулирован в его шаблонном методе `templateMethod()`. Ключевое слово `final` указывает, что этот метод фиксирован и не может быть замещен в подклассах данного класса. Но это означает лишь то, что в подклассах не может быть изменен *порядок следования операций*. Функции `primitiveA()` и `primitiveB()`, входящие в шаблонный метод, являются абстрактными. Они обязательно требуют конкретного наполнения в подклассах. Поэтому, хотя `templateMethod()` фиксирован, операции, которые в него входят, не обязательно должны быть такими и могут быть замещены в подклассах.

В то же время, если вы хотите зафиксировать какую-либо из операций, составляющих шаблонный метод, вы вполне можете это сделать. Например, функция `fromTemplate()` зафиксирована точно так же, как главный шаблонный метод. Будучи его частью, операция `fromTemplate()` не может быть замещена с помощью ключевого слова `override`, в отличие от двух других функций. Таким образом, вы можете поступать двояко. Те операции, которые вы хотите уточнять в подклассах, вы можете оставлять абстрактными, а те, которые вы хотите оставить постоянными для всех подклассов, вы можете фиксировать с помощью выражения `final`.

В подклассе `ConcreteClass` оба абстрактных метода из абстрактного класса замещаются и имплементируются. Подкласс дает им совсем другое содержимое. Но той операции, которая была зафиксирована, `fromTemplate()`, вы не найдете, поскольку она наследуется из абстрактного класса, и ее нельзя изменить в подклассе. Больше здесь нечего добавить.

Тестирование шаблонов

Чтобы протестировать наш минималистский пример Шаблонного метода, создайте представителя конкретного класса и вызовите у него шаблонный метод. Как вы видели, в конкретном классе две операции изменяются и имплементируются, но больше ничего не делается. Сохраните пример 9.3 в том же каталоге, что и другие два файла минималистского примера, используя имя из заголовка примера.

Пример 9.3. TestTemplate.as

```
package
{
    import flash.display.Sprite;
    public class TestTemplate extends Sprite
    {
        public function TestTemplate()
        {
            var testPlate:AbstractClass=new ConcreteClass();
            testPlate.templateMethod();
        }
    }
}
```

Чтобы протестировать наш пример, откройте новый файл документа Flash и введите `TestTemplate` в поле его класса документа на панели свойств. После запуска этого приложения вы должны увидеть следующий результат на вашей панели `Output`:

```
Concrete class
Special A
Special B
Hello everybody!
```

Данный результат показывает, что детали реализации, добавленные подклассом в дополнение к инвариантному методу, вызываются в фиксированном порядке алгоритма шаблонного метода. Поскольку главной задачей этого примера было показать вам структуру Шаблонного метода, вам легко увидеть в нем, как некоторые детали изменяются в подклассе (строки `Special A` и `Special B`), как сохраняется порядок исполнения основного алгоритма, и как работают постоянные (фиксированные) операции.

Использование гибкости Шаблонного метода

Теперь, когда вы уже лучше понимаете как работает структура Шаблонного метода, давайте посмотрим, как она может быть продуктивно применена. В следующем примере вы увидите имплементацию Шаблонного метода, в котором один шаблонный алгоритм используется для двух различных конкретных классов. Рисунок 9.4 показывает его диаграмму классов:

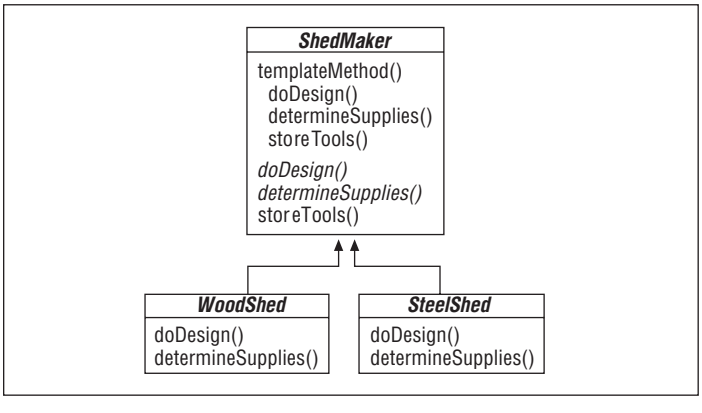


***Рис. 9.4.** Шаблонный метод с двумя конкретными классами*

Основной абстрактный класс данного примера `ShedMaker` («строитель сараев») содержит три операции в своем шаблонном методе `templateMethod()`. Две из них являются абстрактными и будут замещаться и наполняться конкретными действиями в подклассах. Третья операция – фиксированная функция, которая не замещается и будет использоваться во всех подклассах без изменений.

В данном примере рассматривается последовательность действий по строительству небольшого сарая для инструментов, а также возможные различные типы сараев. Мы разработаем алгоритм такой постройки, который будет включать в себя следующее:

- Проектирование сарая
- Определение необходимых материалов

- Размещение инструментов в готовом сарае

Эти шаги используются при строительстве любого сарая, но в зависимости от его типа их конкретные детали могут несколько различаться. В нашем случае мы считаем, что есть два варианта строительства сарая: из дерева и из гофрированного железа. Поскольку проектирование сарая из дерева отличается от проектирования сарая из металла, данная операция (`doDesign()`) является абстрактной. И определение состава требуемых для строительства материалов будет различным для этих двух случаев, поэтому функция `determineSupplies()` также является абстрактной. Однако независимо от того, какой именно сарай будет построен, размещение в нем инструментов будет одинаковым, поэтому данная функция является фиксированной и не будет меняться.

Для начала откройте три новых файла ActionScript 3.0 и введите в них код из примеров с 9.4 по 9.6. Сохраните их в одном каталоге, используя имена файлов из заголовков примеров.

Пример 9.4. ShedMaker.as

```
package
{
    // Абстрактный класс
    class ShedMaker
    {
        public final function templateMethod():void
        {
            doDesign();
            determineSupplies();
            storeTools();
        }
        protected function doDesign():void
        {
        }
        protected function determineSupplies():void
        {
        }
        private final function storeTools():void
        {
            trace("Now I can put all my tools away.\n");
        }
    }
}
```

Пример 9.5. WoodShed.as

```
package
{
    class WoodShed extends ShedMaker
    {
        trace("WoodShed");
        override protected function doDesign():void
```

```
        {
            trace("Designing Wood Shed");
        }
        override protected function determineSupplies():void
        {
            trace("I'll do it with 1 X 12's for the walls.");
            trace("The rest I'll do with 2 X 4's.");
        }
    }
}
```

Пример 9.6. SteelShed.as

```
package
{
    class SteelShed extends ShedMaker
    {
        trace("SteelShed");
        override protected function doDesign():void
        {
            trace("Designing Steel Shed");
        }
        override protected function determineSupplies():void
        {
            trace("Ok I'll need some corregated sheet metal.")
            trace("Better get some steel fasteners too.");
        }
    }
}
```

В обоих этих конкретных подклассах используется выражение `trace()` для отображения на панели Output, какой именно их представитель создается в приложении. Каждый из двух конкретных классов добавляет уникальные действия в две абстрактные операции `doDesign()` и `determineSupplies()`. Но для функции `storeTools()` от них ничего не нужно, поскольку она инвариантна для всех имплементаций. Она наследуется прямо из класса `ShedMaker`.

Чтобы все протестировать в главном классе приложения `BuildShed` («строим сарай»), создаются представители обоих конкретных классов. В примере 9.7 показано, что для создания двух различных объектов `wood` и `steel` используется тип интерфейса. Сохраните этот пример в файле с именем из его заголовка в том же каталоге, что и примеры с 9.4 по 9.6.

Пример 9.7. BuildShed.as

```
package
{
    import flash.display.Sprite;
    public class BuildShed extends Sprite
    {
        public function BuildShed()
```

```
        {
            // Создаем стальной сарай
            var steel:ShedMaker=new SteelShed();
            steel.templateMethod();

            // Создаем деревянный сарай
            var wood:ShedMaker=new WoodShed();
            wood.templateMethod();
        }
    }
}
```

Оба объекта тестового класса задействуют шаблонный метод `templateMethod()`, но, как вы увидите при тестировании данного приложения, результаты его применения будут существенно различаться. Откройте новый файл документа Flash и в его поле класс документа введите `BuildShed`. Когда вы запустите это приложение, вы увидите следующие результаты его работы в окне Output:

```
SteelShed
Designing Steel Shed
Ok I'll need some corregated sheet metal.
Better get some steel fasteners too.
Now I can put all my tools away.

WoodShed
Designing Wood Shed
I'll do it with 1 X 12's for the walls.
The rest I'll do with 2 X 4's.
Now I can put all my tools away
```

Здесь вы видите результаты работы Шаблонного метода. Для первого объекта были вызваны операции конкретного класса `SteelShed`. Все результаты их работы относятся к строительству металлического сарая, за исключением последней строки перед строкой «`WoodShed`». Эта последняя строка является результатом фиксированной функции из абстрактного класса `ShedMaker`. Затем другой объект `WoodShed` получил совершенно другой набор результатов от вызова функции `templateMethod()`, за исключением последней строки.

Мы надеемся, что структура Шаблонного метода станет вам гораздо понятнее благодаря этому примеру. Все вызовы операций совершаются от класса более высокого уровня `ShedMaker`, из его шаблонного метода `templateMethod()`, к более низкоуровневым конкретным классам. Поэтому, хотя подклассы замещают функции и определяют действия для некоторых операций в шаблонном методе, реальные вызовы происходят согласно Голливудскому принципу сверху вниз из шаблонного метода.

Выбор и воспроизведение звука и видео

Рассматривая то, как Flash работает со звуком и видео, можно отметить, что в этих процессах есть много общего. Например, им обоим требуется файл для воспроизведения, и у обоих есть что-то вроде команды «воспроизвести». Используя Шаблонный метод, мы сможем закрепить такие две операции получения имени файла и воспроизведения в некотором обобщенном алгоритме. Однако ввиду очевидного различия между звуком и видео нам придется оставить все детали этих операций на подклассы. Фиксированной операцией в нашем главном абстрактном классе будет текстовый заголовок.

Общий формат

Наш начальный абстрактный класс, содержащий общий алгоритм выбора имени файла и команды на воспроизведение, будет весьма прост, поскольку эти операции будут без конкретного содержания. Третья его операция, которая добавляет текстовый заголовок в верхней части экрана, будет фиксированной. Хотя она относительно проста, она потребует импорта ряда пакетов. Откройте новый файл ActionScript, скопируйте в него содержимое примера 9.8 и сохраните его, используя заголовок этого примера в качестве имени файла.

Пример 9.8. VidAudio.as

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    import flash.text.TextFormat;

    // Абстрактный класс
    class VidAudio extends Sprite
    {
        private var mText:TextField;

        // Шаблонный метод
        public final function mediaProducer ():void
        {
            selectMedia ();
            playNow ();
            fromMediaDesign ();
        }
        protected function selectMedia ():void
        {
            // для замещения
        }
```

```
        protected function playNow ():void
        {
            // для замещения
        }
        private final function fromMediaDesign ():void
        {
            mText=new TextField();
            mText.autoSize=TextFieldAutoSize.CENTER;
            mText.background=false;
            mText.text="Welcome to Template Media!";
            addChild (mText);
            var pos:Number=mText.length;
            mText.x= 200-(pos/2);
            mText.y=10;
        }
    }
}
```

Здесь шаблонный метод называется `mediaProducer()` **(«медиапродюсер»),** он фиксирует порядок следования трех своих операций: `selectMedia()` **(«выбор»),** `playNow()` **(«воспроизведение»)** и `fromMediaDesign()` **(«надпись от дизайнера»).** В последующих конкретных подклассах операциям выбора и воспроизведения будут добавлены действия, а функция `fromMediaDesign()` уже полностью готова к употреблению и не нуждается в каких-либо уточнениях.

Не очень конкретные конкретные классы

Следующим шагом нам надо добавить парочку конкретных классов для работы с видео и аудио. Во-первых, им обоим нужно выбрать файлы; но вместо того чтобы выполнять реалистичную операцию выбора файлов динамически, оба наших конкретных класса просто выдают фиксированные имена. Это довольно просто, но хорошо иллюстрирует концепцию добавления необходимых действий в операции. Не стесняйтесь изменить данную операцию, чтобы она принимала нормальный динамический ввод. Операции воспроизведения создают все необходимое для проигрывания того, что было передано в текстовой строке в качестве имени файла. Откройте еще два файла ActionScript и поместите примеры 9.9 и 9.10 в тот же каталог, что и предыдущий абстрактный класс, используя имена из заголовков в качестве имен этих файлов.

Пример 9.9. Vid.as

```
package
{
    // Конкретный класс
    // Класс Vid
    public class Vid extends VidAudio
    {
        private var vidName:String;
```

```
        override protected function selectMedia ():void
        {
            vidName="media";
        }
        override protected function playNow ():void
        {
            var playVideo=new PlayVideo(vidName);
            this.addChild (playVideo);
        }
    }
}
```

Пример 9.10. Audio.as

```
package
{
    // Конкретный класс
    // Класс Audio
    public class Audio extends VidAudio
    {
        private var tuneName:String;
        override protected function selectMedia ():void
        {
            tuneName="iBlues";
        }
        override protected function playNow ():void
        {
            var playTune:PlayTune=new PlayTune(tuneName);
        }
    }
}
```

Здесь процесс воспроизведения аудио или видео отдан специализированным классам, которые занимаются всеми деталями. То есть операции, проигрывающие видео и аудио, помещены в другие классы, и все, что нам нужно сделать в наших конкретных классах, – это создать представителей соответствующих классов и добавить их в список отображения.

В этой точке вам надо обратить внимание на связь между нашими двумя операциями. Порядок их следования в алгоритме не изменился, однако они соединены общей приватной строковой переменной, хранящей имя файла. Заметьте, что оба наших конкретных класса используют функции `selectMedia()` и `playNow()`, хотя они совершенно разные.

Детальные классы

Хотя Шаблонный метод позволяет добавлять детальные действия своим операциям с помощью конкретных подклассов, операции для воспроизведения видео и аудио требуют еще большей детализации. Поэтому они не помещены в конкретные классы, а имеют собственные спе-

циализированные классы. Конечно, таким образом создаются классы, которые вы сможете, весьма вероятно, многократно использовать и в других приложениях.

Класс для видео довольно замысловатый, а класс для звука совершенно простой. Создайте два новых файла ActionScript и сохраните в них код из примеров 9.11 и 9.12, используя заголовки в качестве имен этих файлов. Проследите, чтобы они оказались в том же каталоге, что и остальные файлы данного приложения.

Пример 9.11. PlayVideo.as

```
package
{
    import flash.display.Sprite;
    import flash.media.Video;
    import flash.events.NetStatusEvent;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.text.TextField;

    public class PlayVideo extends Sprite
    {
        private var vidNow:Video;
        private var durText:TextField;

        public function PlayVideo(vid:String)
        {
            var nc:NetConnection=new NetConnection();
            nc.connect(null);
            var ns:NetStream=new NetStream(nc);
            vidNow=new Video();
            vidNow.attachNetStream(ns);
            vid+=".flv";
            ns.play(vid);
            vidNow.x=((550/2)-(vidNow.width/2));
            vidNow.y=50;
            addChild(vidNow);
            ns.addEventListener(NetStatusEvent.NET_STATUS, stopClear);

            // проверка MetaData
            var dummy:Object=new Object();
            ns.client=dummy;
            dummy.onMetaData=getMeta;
        }

        // Очистить видео и текст длительности, когда видео закончится
        function stopClear(event:NetStatusEvent):void
        {
```

```
            if (event.info.code=="NetStream.Play.Stop")
            {
                vidNow.clear();
                durText.text="";
            }
        }

        // Показать
        function getMeta(mdata:Object):void
        {
            durText=new TextField();
            this.addChild(durText);
            durText.text="Duration:"+mdata.duration;
            durText.x=((550/2)-(durText.width/2));
            durText.y=300;
        }
    }
}
```

Пример 9.12. PlayTune.as

```
package
{
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;

    public class PlayTune
    {
        private var channelNow:SoundChannel;

        public function PlayTune(tuneNow:String)
        {
            tuneNow+=".mp3";
            var playMe:URLRequest=new URLRequest(tuneNow);
            var soundMedia:Sound=new Sound();
            soundMedia.load(playMe);
            channelNow=soundMedia.play();
        }
    }
}
```

В ходе разработки класса для видео мы не только создали триггер для воспроизведения, но и добавили функции по очистке экрана по его завершении и извлечения информации о продолжительности видео. Все это делает данный класс гораздо более удобным для применения в других приложениях. В классах для видео и аудио к именам файлов для воспроизведения добавляются соответствующие расширения, так что для использования этих классов необходимо лишь имя файла без расширения.

Воспроизведение

Наконец, нашему приложению нужен интерфейс пользователя для выбора между звуком и видео. В нем требуются функции вызова каждого из двух конкретных классов шаблонного метода и несколько функций для необходимых кнопок. Откройте новый файл ActionScript и введите в него код из примера 9.13. Сохраните этот файл, используя заголовок примера, в том же каталоге, где находятся остальные файлы для данного приложения.

Пример 9.13. PlayMedia.as

```
package
{
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.events.MouseEvent;

    public class PlayMedia extends Sprite
    {
        var videoButton:VideoButton;
        var tuneButton:TuneButton;

        public function PlayMedia ():void
        {
            doButton ();
        }
        // Берем кнопки из библиотеки и добавляем на сцену
        private function doButton ():void
        {
            tuneButton=new TuneButton();
            videoButton=new VideoButton();
            addChild (tuneButton);
            addChild (videoButton);
            tuneButton.x=((stage.stageWidth/2)-
                (1.5*tuneButton.width)), tuneButton.y=30;
            videoButton.x=((stage.stageWidth/2)+5), videoButton.y=30;
            tuneButton.addEventListener
                (MouseEvent.CLICK,getTune,false,0,true);
            videoButton.addEventListener
                (MouseEvent.CLICK,getVideo,false,0,true);
        }

        // Вызываем шаблонный метод для видео
        private function getVideo (e:MouseEvent):void
        {
            var vidUp:VidAudio=new Vid();
            vidUp.mediaProducer ();
            addChild (vidUp);
        }
        // Вызываем шаблонный метод для аудио
```

```
        private function getTune (e:MouseEvent):void
        {
            var tuneUp:VidAudio=new Audio();
            tuneUp.mediaProducer ();
            addChild (tuneUp);
        }
    }
}
```

Большая часть работы в данном приложении проделана с помощью ActionScript, однако документ Flash будет использован в нем для создания кнопок пользовательского интерфейса. Следующие шаги показывают как именно.

1. Откройте новый документ Flash и сохраните его в каталоге с другими файлами данного приложения под именем *PlayMedia.fla*.
2. Выберите из меню Insert → New Symbol.
3. В диалоговом окне New Symbol выберите в качестве типа Button. Введите VideoButton в поле Name и отметьте позицию выбора Export for ActionScript. Щелкните на OK, чтобы перейти в режим редактирования символа.
4. Выберите инструмент для рисования прямоугольника и установите радиус закругления его углов в 8. Нарисуйте прямоугольник шириной в 49, а высотой в 60 пикселов.
5. Добавьте второй уровень, продолжая находиться в режиме редактирования символа. Назовите верхний уровень Text, а нижний уровень – Shape. На текстовом уровне добавьте статическую текстовую метку Video. Щелкните на OK.
6. Повторите шаги со 2-го по 5-ый, изменив имя кнопки на TuneButton и текстовую метку из шага 5 на Tune.
7. На вашей панели Library теперь должны быть две кнопки с именами VideoButton и TuneButton. Каждая из них является классом, и объекты каждого из этих классов используются в классе PlayMedia.

Для работы этого приложения вам потребуются два дополнительных файла: *mp3*-файл и *flv*-файл. Переименуйте *mp3*-файл в *iBlues.mp3*, а *flv*-файл – в *media.flv*. Рисунок 9.5 показывает вид нашего приложения при воспроизведении видео.

Как вы видели в данном примере, детализация операций основного алгоритма может быть весьма далеко идущей. Рассматривая наши два последних конкретных класса, легко отметить, что методы для оперирования с видео более проработанные, чем для аудио. Они не только включают в себя запуск воспроизведения, но и извлекают метаданные из *flv*-файла и показывают их в текстовом поле на сцене. Что касается звука, то для его воспроизведения даже не используется добавление объектов класса для аудио в список отображения, поскольку звук не отображается на сцене.

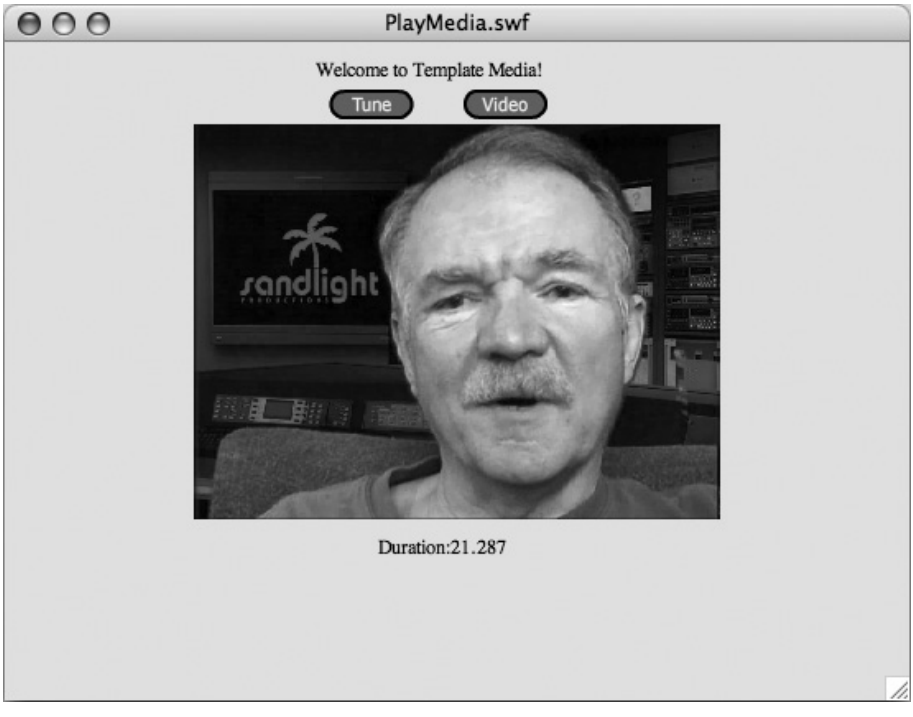


Рис. 9.5. *Приложение для воспроизведения видео и аудио*

Подцепляй

Шаблонный метод используется также для контроля расширений основного алгоритма в подклассах. При обсуждении области применения Шаблонного метода разработчики концепции шаблонов проектирования («банда четырех») поясняют, что они называют подключаемой операцией. Подключаемая операция размещается в шаблонном методе, чтобы его возможные расширения могли «прицепиться» к главному алгоритму.

Для программистов на ActionScript 3.0 подключаемые функции будут выглядеть точно так же, как абстрактные функции, поскольку в этом языке нет настоящих абстрактных функций. Таким образом, нам нужно все время помнить о разнице между ними, заключающейся в том, что подключаемые операции *могут* быть замещены, а абстрактные операции *обязательно должны* быть замещены и имплементированы. Для обозначения этих функций очень полезно использовать соответствующие комментарии, как в следующем фрагменте сценария:

```
// Абстрактная функция
function doAbstract():void {}
```

```
// Подключаемая функция
function doHook():void {}
```

Ни абстрактная функция, ни подключаемая функция не обязана содержать какие-либо конкретные действия в своем начальном описании. Однако предназначение у них разное, и наличие начального функционала у подключаемой операции позволяет обрабатывать ситуации по умолчанию.

Когда цеплять?

Подключаемые операции очень полезны когда некоторые части вашего алгоритма *только иногда* нуждаются в детализации со стороны подклассов. Поскольку подклассы должны обеспечивать такую детализацию для операций алгоритма шаблонного метода, подключаемые операции требуются вам не так часто, как можно было бы ожидать. Подключаемая операция – это конкретный метод, но она может замещаться в подклассах так же, как и любой абстрактный метод; так что нет повода писать ее как конкретный метод. А учитывая, что ActionScript 3.0 вообще не различает абстрактные и конкретные функции, разница между абстрактными и подключаемыми функциями становится еще более размытой.

Одним из способов использования подключаемых операций в шаблонных методах является обслуживание с их помощью нестандартных ситуаций в общем алгоритме. Например, если в 9 из 10 случаев в ваших подклассах подключаемая операция не будет замещена, то ее применение в таком приложении весьма удачно. *Чем меньшее число раз подключаемая операция будет замещена и имплементирована, тем больше от нее пользы*. Данная фраза может показаться парадоксальной, но учтите при размышлении о ней, что подключаемые операции помещают в алгоритм потому, что их замещать необязательно. В большинстве случаев подклассы могут просто имплементировать операции шаблонного метода и на том заканчивать свою работу, используя при этом имплементации по умолчанию для подключаемых операций. Однако если подклассам нужно использовать подключаемую операцию для обработки исключительной ситуации, такая операция становится очень полезной.

Подключаемые операции – скучающие ремонтники

В течение многих лет компания Мэйтег[1] изображала своих ремонтников как скучающих и бездельничающих работников. Смысл такой рекламы заключается в том, что продукция Мэйтэг настолько качественно сделана, что ремонт для нее просто не нужен. Мы можем рассматривать

[1] Maytag Corporation – корпорация, производящая бытовые товары длительного пользования (телевизоры, газовые плиты, холодильники и т. п.). Одна из старейших и наиболее известных фирм такого рода – основана в 1893 г. Ф. Л. Мейтэгом. – *Прим. перев.*

подключаемые операции в шаблонном методе в том же ключе – как скучающих ремонтников, которые очень редко требуются.

Поскольку применение подключаемых операций – это скорее исключение из правил, для их изучения нам нужно рассмотреть сценарий, в котором ситуация практически всегда одинаковая, однако изредка меняется и требует тогда их использования. Одной из самых замечательных зон для полета на самолете является пустынная приграничная область в Южной Калифорнии. Погода там обычно ясная с практически неограниченной видимостью, что позволяет пилотам летать там, следуя правилам визуального полета (ПВП). Полет над полуостровом Байя в Мексике – это прекрасное путешествие, особенно вдоль побережья моря Кортеса. Представьте себе пилота, который возит так туристов к Пунта Буфео на полуострове Байя для посещения лежбищ морских львов. Он каждый день, за редким исключением, может летать по правилам визуального полета и совершать такие рейсы. Однако время от времени погодные условия хуже, чем необходимые для полета по ПВП, – облачно и очень туманно. Поскольку наш пилот имеет ранг, дающий ему право совершать полеты только по ПВП, он не может выполнять полеты по приборам (ППП). Используя подключаемую операцию как зацепку, мы создадим шаблонный метод для данной ситуации.

Полет на Байю с зацепкой

С помощью нашего гипотетического пилота следующее приложение показывает, как использовать подключаемую операцию в Шаблонном методе. Мы полагаем, что пилот совершает свои рейсы из аэропорта Калексико, где погода обычно солнечная и ясная. Откройте четыре новых файла ActionScript 3.0, введите в них код из примеров с 9.14 по 9.17 и сохраните их под именами из заголовков примеров.

Пример 9.14. BajaFlight.as

```
package
{
    // Абстрактный класс
    class BajaFlight
    {
        protected var weatherNow:String;

        public final function templateMethod ():void
        {
            checkAirplane ();
            if (checkWeather()=="Nice and clear")
            {
                fileVFR ();
            }
        }
        protected function fileVFR ():void
        {
```

```
            // для замещения
        }
        protected function checkAirplane ():void
        {
            // для замещения
        }
        // Подключаемая функция
        protected function checkWeather ():String
        {
            weatherNow="Nice and clear";
            return weatherNow;
        }
    }
}
```

Здесь подключаемая функция `checkWeather()` («проверка погоды») по умолчанию возвращает значение «`Nice and clear`» («солнечно и ясно»). Алгоритм шаблонного метода состоит из двух операций: `checkAirplane()` и, при определенных условиях, `fileVFR()`. Обе эти функции абстрактные, и их спецификация оставлена на долю подклассов. Условием для работы операции `fileVFR()` является строка, возвращаемая подключаемым методом `checkWeather()`. Ее реализация по умолчанию делает замещение данной функции обычно ненужным.

Пример 9.15. FlightPlan.as

```
package
{
    // Любой конкретный класс
    public class FlightPlan extends BajaFlight
    {
        trace("*^* Clear as a Bell *^*");
        override protected function checkAirplane():void
        {
            trace("Doing the walk around...looking good.");
        }

        override protected function fileVFR():void
        {
            trace("I'm off to Baja on a beautiful day!");
            trace("First I'll file a VFR flightplan\n");
        }
    }
}
```

Поскольку операции `checkAirplane()` и `fileVFR()` являются абстрактными функциями (по крайней мере по назначению), они должны быть замещены в нашем подклассе. А подключаемая функция считается конкретной, поэтому замещать ее необязательно. В примере 9.15 подключаемый метод `checkWeather()` не включен в новый класс. Он наследуется прямо из класса `BajaFlight` и используется без каких-либо изменений.

Однако в примере 9.16 мы хотим изменить значение, возвращаемое подключаемым методом checkWeather(), поэтому он будет замещен и станет возвращать новое значение «cloudy» («облачно») вместо значения по умолчанию («Nice and clear»). Шаблонный метод вызывает функцию fileVFR(), *только если* checkWeather() возвращает ему значение «Nice and clear». В результате функция fileVFR() никогда не будет выполняться.

Пример 9.16. Cloudy.as

```
package
{
    // Любой конкретный класс
    public class Cloudy extends BajaFlight
    {

        trace ("*^* Cloudy *^*");
        override protected function checkAirplane ():void
        {
            trace ("Doing the walk around...looking good");
        }
        override protected function fileVFR ():void
        {
            trace ("I'm off to Baja on a beautiful day!");
            trace ("First I'll file a VFR flightplan\n");
        }

        // Вызываем подключение
        override protected function checkWeather ():String
        {
            weatherNow="Cloudy";
            return weatherNow;
        }
    }
}
```

В данном примере функция fileVFR() также замещается, для чего используется тот же код, что и в классе FlightPlan. Это было сделано специально, чтобы показать, что совершенно не важно, какой код вы применяете для функции fileVFR() в данном классе. Поскольку подключаемый метод был изменен так, что данная функция не будет выполняться, ее содержимое совершенно не важно.

В отличие от языка Java и других языков, в которых вы можете создавать настоящие абстрактные функции, в ActionScript 3.0 (следуя стандартам ECMAScript) мы вынуждены *притворяться*, что некоторые методы являются абстрактными. Ради такой фантазии метод fileVFR() был замещен в примере 9.16, хотя его можно было вообще не затрагивать в классе Cloudy. Он все равно не будет вызываться, поскольку подключаемый метод запрещает это, изменяя свое возвращаемое значение на что-то отличное от строки «Nice and clear». Так что единственной причиной присутствия ме-

тода в `fileVFR()` в классе `Cloudy` является желание напомнить вам, что абстрактный метод должен быть обязательно замещен в любом подклассе главного абстрактного класса `BajaFlight`.

Мы полагаем, что изучение примеров шаблонов проектирования, в которых абстрактные функции действительно прописаны, будет более легким, если у вас будут собственные «абстрактные» функции в ваших приложениях на ActionScript 3.0. Кроме того, в некоторых шаблонах проектирования, таких как Шаблонный метод, подключаемые методы принципиально отличаются от абстрактных методов, поскольку все абстрактные методы обязательно должны быть замещены, а подключаемые должны быть замещены, только если они были изменены в подклассе.

Наконец у нас все готово к тестированию нашего приложения. В идеале мы должны были бы получать данные о погоде от метеоцентра, которые позволяли бы нам судить, соответствуют условия для полета по ПВП или нет. Вместо этого в тестовом классе `TestFlight` из примера 9.17 просто создаются представители классов `FlightPlan` и `Cloudy`.

Пример 9.17. TestFlight.as

```
package
{
    import flash.display.Sprite;
    public class TestFlight extends Sprite
    {
        public function TestFlight()
        {
            var takeOff:BajaFlight=new FlightPlan();
            takeOff.templateMethod();

            var grounded:BajaFlight=new Cloudy();
            grounded.templateMethod();
        }
    }
}
```

Откройте новый документ Flash и введите `TestFlight` в его поле класс документа. Вы должны увидеть следующие результаты:

```
*^* Clear as a Bell *^*
Doing the walk around...looking good.
I'm off to Baja on a beautiful day!
First I'll file a VFR flightplan

*^* Cloudy *^*
Doing the walk around...looking good
```

В первой их части вы видите, что обе операции алгоритма успешно отработали и выдали свои тексты. Однако в представителе класса `Cloudy` был вызван только метод `checkAirplane()`. Именно это и ожидалось, по-

скольку в классе `Cloudy` подключаемый метод `checkWeather()` возвращает что-то отличное от строки о хорошей погоде, чем отменяет операцию `fileVFR()`.

Заключение

Шаблонный метод показывает нам, как функциональность может быть распределена среди подклассов. Он оставляет большую часть работы по детализации действий на подклассы, но в то же время он может контролировать порядок выполнения операций в общем алгоритме, составляющем его шаблонный метод. Он служит очень хорошим примером контроля взаимозависимостей и использует инвертированную структуру контроля.

Поэтому Шаблонный метод применяется там, где требуются и контроль за действиями, и их изменяемость при общих минимальных взаимозависимостях. Он очень широко используется, и во многих абстрактных классах вы можете найти некоторые элементы его работы. Это действительно общий шаблон проектирования, поскольку он предлагает и контроль алгоритма работы и гибкость подклассов при имплементации методов их класса.

Шаблонный метод весьма полезен и в ActionScript 3.0, даже несмотря на отсутствие в этом языке абстрактных классов и абстрактных функций. Рассмотрение обычных внутренних функций без содержания как абстрактных операций позволяет использовать Шаблонный метод практически идентичным способом, что и в языках, где поддерживаются обе указанные возможности.

Наконец, подключаемые методы обеспечивают Шаблонному методу еще большую гибкость, позволяя вмешиваться в ход исполнения его основного алгоритма. Они работают как служебные входы в этот алгоритм. Все операции шаблонного метода остаются на своих местах, но они могут измениться в смысле того, что они будут делать, или перестать вызываться при работе этого метода. Неудивительно, что Шаблонный метод с его четкой структурой и такой гибкой имплементацией столь широко используется при проектировании приложений.

10

Шаблон Состояние

Государство, которое неспособно
видоизменяться, неспособно и сохраниться.

Эдмунд Берк

Все перевороты Нового времени
приводили к укреплению государства.

Альбер Камю

Шаблон проектирования для создания автоматов

Шаблон проектирования Состояние работает в области различных состояний в приложении, переходов между ними и различного поведения для каждого состояния. Даже в простом приложении управления лампами освещения мы можем обнаружить два состояния: «включено» и «выключено». В выключенном состоянии свет не горит, а во включенном – горит. Переход переключателя света из выключенного состояния во включенное происходит с помощью метода, изменяющего состояние такого приложения, – щелчка по выключателю. Переход из включенного состояние в выключенное происходит с помощью другого метода. Интерфейс содержит в себе все такие переходы, и каждое состояние имплементирует их уникальным для себя образом. Каждый такой метод имплементируется по-своему в зависимости от контекста своего использования. То есть метод включения света `illuminateLight()`, например, будет работать одним способом в выключенном состоянии и совершенно по-другому во включенном, даже хотя он будет входить в оба эти состояния.

Ключевые особенности

Следующие ключевые особенности характеризуют шаблон проектирования Состояние:

- Состояния существуют как внутренние характеристики объекта.
- Объекты меняются определенным образом при смене состояния. Может показаться, что изменяется и класс объекта, но на самом деле меняется лишь его поведение, которое является частью его класса.
- Поведение в каждом состоянии зависит от текущего состояния объекта.

Одним из типов приложений, где шаблон Состояние очень популярен, являются симуляторы различных устройств. Многие устройства, которые меняют состояние объекта, сами изменяются со сменой состояния. Так, ручка громкости радио меняет состояние уровня громкости его звука. Примером более сложного симулируемого устройства может служить панель звукового синтезатора, на которой симулируемые регуляторы изменяют различные свои состояния так, что влияют на работу всего объекта (синтезатора) и на получаемый от него звук. Видеоплеер Flash также имеет несколько состояний клавиш управления: проигрывание, запись, добавление, пауза и стоп. Каждое из состояний видеоплеера ведет себя в зависимости от статуса других состояний, равно как и от своего собственного статуса.

Модель шаблона Состояние

Чтобы понять и оценить значение шаблона проектирования Состояние, мы должны кое-что узнать о *конечных автоматах*. Конечный автомат – это общая теоретическая модель состояний, которые вы будете использовать в своем приложении, применяющем шаблон Состояние. Таким образом, если работа вашего видеоплеера зависит от состояния его клавиш, ваше приложение является конечным автоматом. Кроме того, мы должны упомянуть об автоматных движках, с помощью которых реализуются на практике конечные автоматы. (Вы можете считать чертеж вашего автомобиля аналогом схемы конечного автомата, а сам автомобиль – аналогом автоматного движка, реализующего ее.) Программа, используемая нами для движения от одного состояния к другому, является автоматным движком. Структура данных автоматного движка определяет механизмы обработки внешних сообщений и управления контекстом автомата.

Начнем мы не с наших обычных диаграмм для шаблонов проектирования, а с *диаграмм состояний*. На самом базовом уровне диаграмма состояний – это иллюстрация всех состояний приложения и переходов между ними, и по существу она является моделью для конечного автомата и реализующего его автоматного движка. Возьмем в качестве примера простейшее приложение для видеоплеера, в котором есть только

два состояния: Играть (Play) и Стоп (Stop). Когда приложение запускается в первый раз, оно попадает в состояние Стоп и может только перейти в состояние Играть. Рисунок 10.1 показывает нам соответствующую диаграмму состояний.

Данная иллюстрация нарисована на компьютере. Но, поскольку основная идея использования диаграмм состояний состоит в том, чтобы, начав с грубой исходной идеи, постепенно уточнять ее с помощью серии рисунков, такой процесс пойдет быстрее, если использовать карандаш и бумагу, а не программы для рисования.

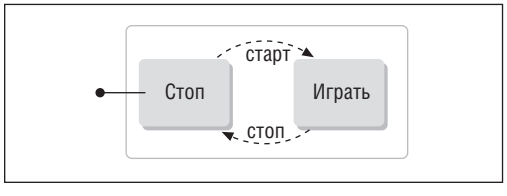


Рис. 10.1. *Простая диаграмма состояний*

Линия, идущая от черной точки к состоянию Стоп, показывает состояние приложения, пока оно не работает, и мы будем полагать, что при запуске оно попадает именно в данное состояние. Это можно было бы проиллюстрировать в виде иерархии состояний работающего и неработающего приложения. Мы могли бы также поместить всю такую иерархию в состояния работающего и выключенного компьютера, но от этого не будет никакой пользы, потому что мы ничего не программируем для этих состояний.

Прежде чем обсуждать переходы из одного состояния в другое, давайте рассмотрим, что каждое из состояний позволяет нам делать. В состоянии Стоп можно только инициировать состояние Играть; остановить воспроизведение нельзя, поскольку оно уже остановлено. А в состоянии Играть можно лишь перейти в состояние Стоп.

Переходы

Переходами в конечных автоматах называют процессы смены состояний. На рис. 10.1 линия, идущая от состояния Стоп к состоянию Играть, может быть методом startPlay() такого процесса. А линия от Играть к Стоп может быть методом stopPlay(). Если бы состояний было больше, то вы, вероятно, обнаружили бы, что не всегда возможно напрямую перейти из одного состояния в другое, а только пройдя через серию промежуточных состояний. Как вы увидите далее в примере, если вы находитесь в состоянии Стоп, то вы не можете напрямую перейти из него в состояние Пауза. Вам нужно сначала перейти в состояние Играть, и только потом вы сможете попасть в состояние Пауза.

Триггеры

Чтобы инициировать переход в другое состояние, вам нужен какой-либо триггер. Триггер – это любое событие, которое запускает переход из состояния в состояние. Обычно под таким событием мы подразумеваем некоторое действие пользователя приложения, вроде движения мыши или нажатия на кнопку, но при симуляции устройств переход в другое состояние может быть вызван и текущей ситуацией, вроде истощения запасов топлива, заряда батареи или столкновения с объектом. Триггеры зависят от контекста ситуации и могут сработать только при определенных условиях. То есть вы можете использовать кнопку `Играть`, чтобы перейти из состояния `Стоп` в состояние `Играть`, но она не будет инициировать состояние `Играть` из самого этого состояния.

Триггеры часто размещают вместе с переходами на диаграмме состояний. Это помогает определять события, запускающие триггер, и соответствующий переход между состояниями. Рисунок 10.2 показывает нашу уточненную диаграмму состояний, содержащую триггеры и переходы, которые они запускают.

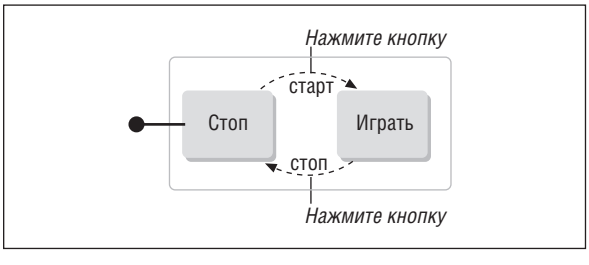


Рис. 10.2. *Диаграмма состояний с переходами и триггерами*

Если вы хотите подробнее изучить тему конечных автоматов, автоматных движков и диаграмм состояний и вопросы реализации работы с состояниями во Flash, то мы рекомендуем вам обратиться к книге «Flash Mx for Interactive Simulation» Джонатана Кайе (Jonathan Kaye) и Дэвида Кастильо (David Castillo) (издательство Thomson, 2003). Хотя в ней и используется Flash, устаревший уже на несколько версий, книга ценна своими концепциями и отличными примерами симуляции некоторых устройств.

Структура шаблона Состояние

Еще в бытность одним из шаблонов проектирования, описанным в книге «банды четырех» «Design Patterns: Elements of Reusable Object-Oriented Software», шаблон Состояние был признан шаблоном, область применения которого вышла за свои первоначальные границы, из области конечных автоматов. Во многом сходный с шаблоном Стратегия шаблон проектирования Состояние может быть использован тогда, когда пове-

дение приложения зависит от изменения его состояний во время работы или имеет сложные условные последовательности операций, зависящие от его текущего состояния. Объект, разработанный с помощью шаблона Состояния, меняет свое поведение при изменении своего внутреннего состояния. Рисунок 10.3 показывает нам общую структуру этого шаблона проектирования с помощью диаграммы классов.

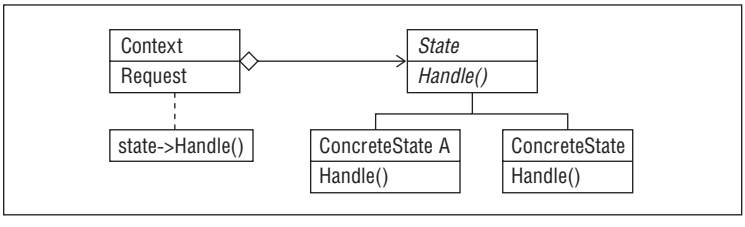


Рис. 10.3. *Шаблон проектирования Состояние*

Ключевые концепции ООП, используемые в шаблоне Состояние

Хотя полиморфизм является основополагающей концепцией ООП, от него не будет пользы, если принципы и задачи его использования поняты не очень хорошо. Полиморфизм – это одна из концепций объектно-ориентированного программирования, применяемая для описания использования различных форм имплементаций объектов. Одной из характеристик полиморфизма, которую вы увидите при изучении шаблона проектирования Состояние, является то, что он достаточно очевиден. Она поможет вам лучше понять, как можно с пользой применять полиморфизм в объектно-ориентированных программах.

Если посмотреть на интерфейс состояния в примере 10.1, то можно увидеть различные методы, которые становятся ядром функциональности для различных состояний. Каждое состояние имеет свой класс. Взглянув на каждый класс, можно заметить, что поведение одних и тех же методов принимает в них различные формы. Налицо работа полиморфизма. Просто посмотрите на различные классы состояний и сравните в них одинаковые по заголовкам методы.

Наверно, одной из самых важных способностей каждого метода состояния является знание своего состояния. Например, в следующих примерах вы увидите, что оба состояния (классы) `Stop` и `Play` имеют у себя метод `startPlay()`. Однако каждый их этих методов ведет себя поразному в своем контексте. Из состояния `Stop` метод `startPlay()` начинает воспроизведение видео. А если его же запустить во время просмотра видео, то он ничего не сделает. Такая пассивность иногда очень важна. Давайте представим, что кто-то смотрел видео и по каким-то причинам нажал на кнопку `Play`. Типичное приложение для воспроизведения ви-

део начало бы при этом заново его показывать. А в этом приложении метод `startPlay()` остается немым, как скала. Полиморфизм позволяет нам иметь в приложении различные формы одного и того же метода, которые знают, как себя вести в различных контекстах. Поэтому в состоянии `Play` рассматриваемый метод знает, что видео уже воспроизводится, и ничего не делает. Пользователь приложения может нажимать на кнопку `Play` сколько угодно, но видео будет просто продолжать показываться. Если пользователь нажмет на кнопку `Stop`, то видео остановится.

Оценить значение полиморфизма поможет приложение, содержащее гораздо больше состояний, за которыми нужно следить. При очень большом количестве методов нам будет крайне сложно исправлять в них какие-либо ошибки. Без полиморфизма мы бы сильно рисковали получить методы с одинаковыми именами, некоторые из которых делали бы не то, что нам нужно. Например, если вы не хотите, чтобы ваше приложение проигрывало видео с самого начала всякий раз при вызове метода `startPlay()`, то, используя шаблон проектирования Состояние, вы можете так организовать ваше приложение, что оно будет показывать видео с начала только в состоянии `Stop`. Вы можете сделать и так, что данный метод будет воспроизводить видео с начала из состояния `Play`. Вы пишете код данного приложения, значит, вы же контролируете поведение всех методов.

В ходе чтения следующей главы о шаблоне проектирования Стратегия у вас может возникнуть ощущение дежавю. Совпадение текстов этих двух глав вовсе не случайно. Когда вы изучите все примеры из каждой главы, вы точно научитесь различать эти шаблоны, несмотря на то что их структуры очень похожи. Шаблон проектирования Состояние специализируется на состояниях и четко определенных переходах. Это одна из причин, почему мы здесь использовали диаграммы состояний; они помогают просто и ясно показать структуру работы данного шаблона, с акцентом на различные состояния и на то, как они переходят друг в друга. Переходы могут контролироваться в шаблоне Состояние либо самими состояниями, либо классом их общего контекста. Кроме того, поскольку шаблон Состояние создает отдельные классы для каждого состояния (среду поведения), в нем обычно создается больше классов, чем в шаблоне Стратегия. Определение, каким именно поведением воспользоваться, делегируется классам состояний, в то время как шаблон Стратегия инкапсулирует в себе набор алгоритмов и позволяет своим клиентам выбирать между ними внутри структуры, которая их использует.

Минималистский абстрактный шаблон Состояние

При использовании шаблона проектирования Состояние все методы (способы поведения) для одного состояния помещаются в один объект (конкретное состояние), а все переходы между состояниями (в конечном автомате) помещаются в один интерфейс. Каждое состояние им-

плементирует данный интерфейс так, как ему это нужно. Ввиду такой структуры для различного поведения, в зависимости от текущего состояния, не требуются никакие условные ветви. Так что нет необходимости писать сложные условные выражения; каждый из объектов состояний сам определяет, как его методы должны вести себя для данного состояния.

Например, для автомата с двумя состояниями `Play` («играть») и `Stop` («стоп») с помощью следующего псевдокода можно определить поведение, начинающее показ видео в зависимости от текущего состояния нашего автомата.

```
function doPlay():void {
    if(state == Play)
    {
        trace("You're already playing.");
    }
    else if (state == Stop)
    {
        trace("Go to the Play state.");
    }
}
```

Когда у нас только пара различных состояний, написать такой код не слишком трудно. Однако при значительном росте общего числа состояний все становится значительно сложнее, и вы можете увидеть просто море подобных условных выражений, работающих совместно. Альтернативой является установка «контекстного» поведения с использованием схемы шаблона Состояние. Пример 10.1 содержит в себе два различных объекта с разной реализацией поведения из одного интерфейса:

Пример 10.1. State.as

```
// Интерфейс
interface State
{
    function startPlay():void;
    function stopPlay():void;
}
// Состояние «проигрывание»
class PlayState implements State
{
    public function startPlay():void
    {
        trace("You're already playing");
    }
    public function stopPlay():void
    {
        trace("Go to the Stop state.");
    }
}
```

```
// Состояние «проигрывание остановлено»
class StopState implements State
{
    public function startPlay():void
    {
        trace("Go to the Play state.");
    }
    public function stopPlay():void
    {
        trace("You're already stopped");
    }
}
```

Как вы можете видеть, поведение (методы) по-разному имплементируется для различных состояний. Если число состояний увеличить, потребуется только добавить новые правила переходов между ними в интерфейсе и создать новое конкретное состояние (класс), которое будет имплементировать свое поведение. Таким образом, каждое новое поведение просто добавляется к существующим классам состояний.

Управление всеми состояниями: работа контекстного класса

Для управления всеми состояниями и переходами между ними вам потребуется какой-то специализированный объект – автоматный движок, реализующий схему вашего конечного автомата. На рис. 10.3 рамка с названием `Context` («контекст») является абстракцией для такого автоматного движка. Этот контекстный класс управляет различными состояниями, создавая тем самым заданный конечный автомат. На рис. 10.4 показана более конкретная диаграмма того, что нам нужно реализовать.

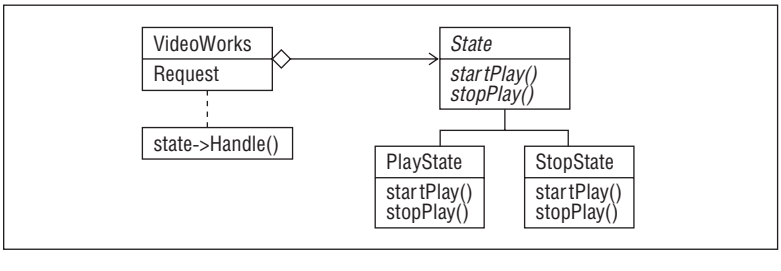


***Рис. 10.4.** Шаблон проектирования Состояние, примененный к видео*

Создание контекстного класса

Для нашего примера реализации простого видеоплеера нам нужен контекстный класс, который будет работать с различными состояниями. Так что сейчас мы им и займемся. Его следует сохранить в файле *Video-*

Works.as. Сначала посмотрите на этот класс в примере 10.2, потом мы обсудим, что в нем происходит.

Пример 10.2. VideoWorks.as

```
1 package
2 {
3     // Контекстный класс
4     class VideoWorks
5     {
6         var playState:State;
7         var stopState:State;
8         var state:State;
9         public function VideoWorks()
10        {
11            trace("Video Player is On");
12            playState = new PlayState(this);
13            stopState = new StopState(this);
14            state=stopState;
15        }
16        public function startPlay():void
17        {
18            state.startPlay();
19        }
20        public function stopPlay():void
21        {
22            state.stopPlay();
23        }
24        public function setState(state:State):void
25        {
26            trace("A new state is set");
27            this.state=state;
28        }
29        public function getState():State
30        {
31            return state;
32        }
33        public function getPlayState():State
34        {
35            return this.playState;
36        }
37        public function getStopState():State
38        {
39            return this.stopState;
40        }
41    }
42 }
```

В начале, в строчках 6–8, данный сценарий создает три объекта класса `State`, по одному для рассмотренных нами состояний (`PlayState` и `StopState`) и еще один (`state`), который будет служить переменной для хране-

ния текущего состояния. Поскольку работа нашего конечного автомата начинается в состоянии `Stop`, переменной `state` присваивается состояние `Stop`. (Это очень похоже на работу выключателя света до того, как вы измените его состояние с «Выключено» на «Включено».)

Далее два метода из интерфейса состояния `State` специфицируются в терминах текущего контекста состояния (строки 16–23). Мы еще создадим код для двух классов состояний, с которыми будет работать этот контекстный класс, но пока можно просто представить себе, что произойдет в этих двух состояниях при вызове их методов. Например, в состоянии `Play` метод `startPlay()` не делает ничего, а в состоянии `Stop` он переключает состояние в `Play`.

В конце добавляются методы получатели и установщики (строки 24–40). Нам потребуется целых шесть методов, по паре этих методов для каждого из трех состояний. Установщики не будут ничего возвращать, а получатели будут возвращать объект `State`.

Завершение и тестирование абстрактного конечного автомата

Чтобы все заработало, нам нужно просмотреть все классы состояний и включить в них ссылки на класс контекста `VideoWorks`. Сохраните пример 10.3 под именем *StopState.as*.

Пример 10.3. StopState.as

```
1 package
2 {
3     // Состояние «проигрывание остановлено»
4     class StopState implements State
5     {
6         var videoWorks:VideoWorks;
7         public function StopState(videoWorks:VideoWorks)
8         {
9             trace("--Stop State--");
10            this.videoWorks=videoWorks;
11        }
12        public function startPlay():void
13        {
14            trace("Begin playing");
15            videoWorks.setState(videoWorks.getPlayState());
16        }
17        public function stopPlay():void
18        {
19            trace("You're already stopped");
20        }
21    }
22 }
```

Добавив объект VideoWorks, мы получили доступ к методам установщикам и получателям для каждого состояния. Например, в строке 15 он вызывается для изменения состояния на Play.

Далее мы повторим все то же самое с состоянием Play, как показано в примере 10.4. Сохраните следующий код как *PlayState.as*.

Пример 10.4. PlayState.as

```
1 package
2 {
3     // Состояние «проигрывание»
4     class PlayState implements State
5     {
6         var videoWorks:VideoWorks;
7         public function PlayState(videoWorks:VideoWorks)
8         {
9             trace("--Play State--");
10            this.videoWorks=videoWorks;
11        }
12        public function startPlay():void
13        {
14            trace("You're already playing");
15        }
16        public function stopPlay():void
17        {
18            trace("Stop playing.");
19            videoWorks.setState(videoWorks.getStopState());
20        }
21    }
22 }
```

Чтобы завершить разработку нашего конечного автомата, необходимо создать для него интерфейс, а поскольку у него есть всего два состояния и два действия, сделать это очень просто. Если вернуться к нашей диаграмме состояний для данного примера, на ней можно увидеть только два перехода между состояниями: один начинает показ видео, другой заканчивает его. Поэтому нам нужны только две абстрактные функции для их описания. Сохраните следующий сценарий из примера 10.5 как *State.as*.

Пример 10.5. State.as

```
1 package
2 {
3     // Интерфейс автомата состояний
4     interface State
5     {
6         function startPlay():void;
```

```
7         function stopPlay():void;
8     }
9 }
```

Все действия переходов описаны в строчках 6 и 7. Позднее мы добавим сюда больше переходов по мере роста нашего проекта. Теперь нам нужно создать *fla*-файл с ActionScript, который будет работать как наш конечный автомат.

Чтобы протестировать наше абстрактное приложение и показать возможности шаблона проектирования Состояние, наш тест должен вызывать класс `VideoWorks` и его два состояния `Play` и `Stop`, используя переходы (методы) `startPlay()` и `stopPlay()`. Фактически оба состояния нужно вызвать дважды. Из состояния `Stop` (исходное состояние нашего приложения) нужно сначала перейти в состояние `Play`. Потом нужно вызвать переход еще раз, чтобы убедиться, что состояние понимает новый контекст. То же самое нужно сделать и после перехода обратно в состояние `Stop`. Сохраните пример 10.6 как *TestState.as* в том же каталоге, что и остальные файлы.

Пример 10.6. TestState.as

```
1 package
2 {
3     // Тестируем состояния
4     import flash.display.Sprite;
5     public class TestState extends Sprite
6     {
7         public function TestState():void
8         {
9             var test:VideoWorks = new VideoWorks();
10            test.startPlay();
11            test.startPlay();
12            test.stopPlay();
13            test.stopPlay();
14        }
15    }
16 }
```

Поскольку наше приложение на данной стадии только печатает текстовые сообщения, чтобы их увидеть, вам нужно будет использовать Flash Test (Control → Test or Control → Test Project). Откройте новый документ Flash и в поле класс документа введите `TestState`. Вы должны увидеть следующие результаты работы нашего приложения в окне `Output`:

```
Video Player is On
--Play State--
--Stop State--
Begin playing
A new state is set
You're already playing
```

```
Stop playing.
A new state is set
You're already stopped
```

Поскольку классы VideoWorks, PlayState и StopState содержат в себе выражения trace() для индикации создания их представителей, они проявляются сразу после начала тестирования нашего приложения. Так как начальным состоянием приложения является Stop, оно меняется на Play при первом вызове метода startPlay(). В связи с тем что вы меняете состояние, выражение trace() из класса VideoWorks сообщает о таком изменении. Когда второй метод startPlay() вызывается второй раз, этот же метод в новом контексте понимает, что он уже находится в состоянии Play и просто сообщает об этом. При нажатии кнопки Stop вы перемещаетесь в состояние Stop, но при повторном ее нажатии та же ее функция понимает, что вы уже находитесь в состоянии Stop, и просто сообщает об этом факте.

Настоящий видеоплеер с состояниями

Пока что в качестве результатов вы видели только сообщения от функции trace(), которые позволяли судить, как работают шаблон проектирования Состояние и наш конечный автомат. Чтобы выводилось что-то более интересное, нам надо включить в наше приложение объект NetStream и строку с названием *flv*-файла. Эта строка нужна нам только при проигрывании видео, поскольку мы можем его остановить, просто закрыв представителя NetStream. Следующие четыре сценария позволяют нашему конечному автомату реально воспроизводить видео и останавливать его показ. Все выражения trace() оставлены в них на месте.

Для имплементации приложения, способного действительно показывать видео, нам необходимо импортировать все необходимые для этого части. Поскольку класс NetStream используется в интерфейсе и в двух классах состояний, каждый из этих файлов нуждается в импорте этого класса. Однако, хотя класс VideoWorks использует оба класса состояний, ему класс NetStream для работы не нужен, поскольку он уже импортирован в эти классы.

Следующие пять листингов кода из примеров с 10.7 по 10.11 следует ввести в файлы ActionScript и сохранить под именами из их заголовков в одном каталоге.

Пример 10.7. State.as

```
package
{
    // Интерфейс состояния
    import flash.net.NetStream;
    interface State
    {
```

```
        function startPlay(ns:NetStream,flv:String):void;
        function stopPlay(ns:NetStream):void;
    }
}
```

В следующем классе `StopState`, показанном в примере 10.8, вы заметите, что метод `startPlay()` имплементирован так, чтобы действительно воспроизводился указанный ему *flv*-файл. Такой переход к состоянию `Play` означает не начало показа видео, а скорее установку такого состояния приложения, в котором происходит воспроизведение видео.

Пример 10.8. StopState.as

```
package
{
    // Состояние «проигрывание остановлено»
    import flash.net.NetStream;

    class StopState implements State
    {
        private var videoWorks:VideoWorks;
        public function StopState(videoWorks:VideoWorks)
        {
            trace("--Stop State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            ns.play(flv);
            trace("Begin playing");
            videoWorks.setState(videoWorks.getPlayState());
        }
        public function stopPlay(ns:NetStream):void
        {
            trace("You're already stopped");
        }
    }
}
```

Обратите внимание, что в классе `PlayState` из следующего примера 10.9 метод `startPlay()` разумно ничего не делает, кроме печати сообщения пользователю, что видео уже показывается. В режиме тестирования приложения это сообщение появлялось в окне `Output`. Но в обычном режиме работы такое сообщение не показывается; продолжение показа видео говорит само за себя.

Пример 10.9. PlayState.as

```
package
{
    // Состояние «проигрывание»
```

```
    import flash.net.NetStream;

    class PlayState implements State
    {
        private var videoWorks:VideoWorks;
        public function PlayState(videoWorks:VideoWorks)
        {
            trace("--Play State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You're already playing");
        }
        public function stopPlay(ns:NetStream):void
        {
            ns.close();
            trace("Stop playing.");
            videoWorks.setState(videoWorks.getStopState());
        }
    }
}
```

Контекстный класс из примера 10.10 собирает для данного шаблона проектирования все вместе. Обратите внимание, что в нем нет импорта объекта `NetStream`**. Однако его листинг ясно показывает, что такой объект является одним из параметров для функций** `startPlay()` **и** `stopPlay()`**. Если вы посмотрите внимательнее, то увидите, что обе эти функции берутся из классов** `PlayState` **и** `StopState`**, в которых класс** `NetStream` **уже был импортирован.**

Пример 10.10. VideoWorks.as

```
package
{
    import flash.net.NetStream;
    // Контекстный класс
    class VideoWorks
    {
        private var playState:State;
        private var stopState:State;
        private var state:State;
        public function VideoWorks()
        {
            trace("Video Player is on");
            playState = new PlayState(this);
            stopState = new StopState(this);
            state=stopState;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
```

```
            state.startPlay(ns,flv);
        }
        public function stopPlay(ns:NetStream):void
        {
            state.stopPlay(ns);
        }
        public function setState(state:State):void
        {
            trace("A new state is set");
            this.state=state;
        }
        public function getState():State
        {
            return state;
        }
        public function getPlayState():State
        {
            return this.playState;
        }
        public function getStopState():State
        {
            return this.stopState;
        }
    }
}
```

В дополнение к классам, входящим в шаблон проектирования Состояние, вам нужны классы кнопок `NetBtn` и `BtnState` из примеров 10.11 и 10.12. На самом деле файлы с этими классами потребуются вам для всех примеров из данной главы, причем должны находиться в тех же каталогах, что и остальные файлы. Мы не включили данные файлы в другие листинги примеров данной главы, потому что они абсолютно одинаковы для всех примеров. (Значит, если вы ввели их один раз, то больше с ними ничего делать не нужно.) Поэтому держите эти файлы под рукой и следите, чтобы они были в каталогах с вашими остальными классами.

Пример 10.11. NetBtn.as

```
package
{
    // Кнопка для переключения триггеров
    import flash.display.Sprite;
    import flash.display.SimpleButton;
    import flash.display.Shape;
    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    public class NetBtn extends SimpleButton
    {
```

```
        public function NetBtn (txt:String)
        {
            upState = new BtnState(0xfab383, 0x9e0039,txt);
            downState = new BtnState(0xffffff,0x9e0039, txt);
            overState= new BtnState (0x9e0039,0xfab383,txt);
            hitTestState=upState;
        }
    }
}
```

Пример 10.12. BtnState.as

```
package
{
    // Состояния для кнопок переключения
    import flash.display.Sprite;
    import flash.display.Shape;
    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    class BtnState extends Sprite
    {
        public var btnLabel:TextField;
        public function BtnState (color:uint,color2:uint,btnLabelText:String)
        {
            btnLabel=new TextField  ;
            btnLabel.text=btnLabelText;
            btnLabel.x=5;
            btnLabel.autoSize=TextFieldAutoSize.LEFT;
            var format:TextFormat=new TextFormat("Verdana");
            format.size=12;
            btnLabel.setTextFormat (format);
            var btnWidth:Number=btnLabel.textWidth + 10;
            var bkground:Shape=new Shape;
            bkground.graphics.beginFill (color);
            bkground.graphics.lineStyle (2,color2);
            bkground.graphics.drawRect (0,0,btnWidth,18);
            addChild (bkground);
            addChild (btnLabel);
        }
    }
}
```

Для тестирования нашего приложения вам потребуется *flv*-файл с именем *test.flv*. Вы можете конвертировать в него любой существующий видеофайл (например, *avi*-файл, *mov*-файл) или использовать любой другой *flv*-файл, изменив его имя. Поместите этот файл в тот же каталоге, что и остальные файлы. Наконец, вам потребуется сценарий для тестирования вашего приложения. Поэтому откройте новый файл ActionScript, введите в него код из листинга примера 10.13 и сохраните его как *TestVid.as*.

Пример 10.13. TestVid.as

```
package
{
    // Имплементация приложения FMS2 и тестирование автомата состояний
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.text.TextField;
    import flash.text.TextFieldType;
    import flash.events.MouseEvent;
    import flash.events.NetStatusEvent;

    public class TestVid extends Sprite
    {
        private var nc:NetConnection=new NetConnection();
        private var ns:NetStream;
        private var vid:Video=new Video(320,240);
        private var vidTest:VideoWorks;
        private var playBtn:NetBtn;
        private var stopBtn:NetBtn;
        private var flv:String;
        private var flv_txt:TextField;
        private var dummy:Object;

        public function TestVid ()
        {
            nc.connect (null);
            ns=new NetStream(nc);
            addChild (vid);
            vid.x=(stage.stageWidth/2)-(vid.width/2);
            vid.y=(stage.stageHeight/2)-(vid.height/2);

            // Создаем автомат состояний
            vidTest=new VideoWorks();

            // Кнопки проигрывания и остановки
            playBtn=new NetBtn("Play");
            addChild (playBtn);
            playBtn.x=(stage.stageWidth/2)-50;
            playBtn.y=350;
            stopBtn=new NetBtn("Stop");
            addChild (stopBtn);
            stopBtn.x=(stage.stageWidth/2)+50;
            stopBtn.y=350;

            // Добавляем обработчики событий
            playBtn.addEventListener (MouseEvent.CLICK,doPlay);
            stopBtn.addEventListener (MouseEvent.CLICK,doStop);
```

```
        // Добавляем текстовое поле
        flv_txt= new TextField();
        flv_txt.border=true;
        flv_txt.borderColor=0x9e0039;
        flv_txt.background=true;
        flv_txt.backgroundColor=0xfab383;
        flv_txt.type=TextFieldType.INPUT;
        flv_txt.x=(stage.stageWidth/2)-45;
        flv_txt.y=10;
        flv_txt.width=90;
        flv_txt.height=16;
        addChild (flv_txt);

        // Для предотвращения ошибки MetaData
        dummy=new Object();
        ns.client=dummy;
        dummy.onMetaData=getMeta;

        // NetStream
        ns.addEventListener (NetStatusEvent.NET_STATUS, flvCheck);
    }
    // MetaData
    private function getMeta (mdata:Object):void
    {
        trace (mdata.duration);
    }
    // Работаем с flv
    private function flvCheck (event:NetStatusEvent):void
    {
        switch (event.info.code)
        {
            case "NetStream.Play.Stop" :
                vidTest.stopPlay (ns);
                vid.clear ();
                break;
            case "NetStream.Play.StreamNotFound" :
                vidTest.stopPlay (ns);
                flv_txt.text="File not found";
                break;
        }
    }
    // Начинаем проигрывание
    private function doPlay (e:MouseEvent):void
    {
        if (flv_txt.text != "" && flv_txt.text !=
            "Provide file name")
        {
            flv_txt.textColor=0x000000;
            flv=flv_txt.text + ".flv";
            vidTest.startPlay (ns,flv);
```

```
                vid.attachNetStream (ns);
            }
            else
            {
                flv_txt.textColor=0xcc0000;
                flv_txt.text="Provide file name";
            }
        }
        // Останавливаем проигрывание
        private function doStop (e:MouseEvent):void
        {
            vidTest.stopPlay (ns);
            vid.clear ();
        }
    }
}
```

Откройте новый документ Flash, в центре его сцены нарисуйте прямоугольник размерами 320 (ширина) на 240 (высота) линией толщиной в 6 пунктов цветом 9E0039 и заполните его цветом FAB383. Используйте панель выравнивания объектов (Align), чтобы ваш прямоугольник был точно посередине, поскольку он будет служить фоном для видео, которое будет показываться поверх него. В поле класс документа введите `TestVid` и протестируйте данное приложение.

Наш интерфейс пользователя крайне прост и связан с переходами между `Start` и `Stop` при показе видео. Вы сможете увидеть связь между показом видео и сообщениями от выражений `trace()`, которые будут показывать, что происходит при нажатии на кнопки. Например, если вы нажмете на кнопку `Start` при показе видео, то ничего не произойдет, потому что в состоянии `Play` функция `startPlay()` ничего не сделает, кроме выдачи сообщения `trace()`, что вы уже смотрите видео.

Расширение шаблона Состояние: добавление новых состояний

Основополагающей особенностью практически всех шаблонов проектирования является их способность к расширению и восприятию изменений. Тип изменений, которые вы предполагаете в будущем для своего приложения, в какой-то степени влияет на выбор для него шаблона проектирования. В нашем конкретном приложении мы добавим в него еще одно состояние.

Добавление состояния Пауза в диаграмму состояний

Первым состоянием, которое мы добавим в наш конечный автомат, будет состояние паузы (`Pause`). Это состояние существует только внутри состояния показа видео (`Play`), и вы не можете в него попасть из состоя-

ния стоп (Stop). Чтобы попасть в состояние Pause, вы должны сначала попасть в состояние Play, а потом вы сможете включать и выключать состояние Pause. Чтобы корректно описать это новое состояние, нам нужно использовать иерархическую диаграмму состояний. Рисунок 10.5 показывает такую диаграмму с необходимой иерархией.

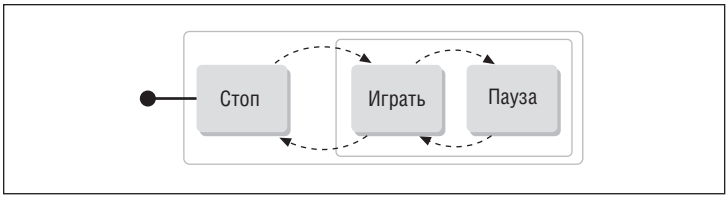


***Рис. 10.5.** Иерархическая диаграмма состояний*

Иерархия на рис. 10.5 очень проста. Ее первый уровень составляют состояния Play и Stop, а внутри состояния Play есть состояния Pause и NoPause.

Поскольку функция паузы производит переключения между состояниями Play и Pause, состояние NoPause в точности равно состоянию Play. Поэтому, вместо того чтобы создавать функции «начать_паузу» и «закончить_паузу», мы просто заставим метод «пауза» вести себя по-разному в различных состояниях. В состоянии паузы он будет возвращать приложение в состояние Play, а в состоянии Play он переведет его в состояние Pause.

Добавление нового поведения

Первым делом мы добавим новое поведение (метод) к уже существующим в файле *State.as*. Это будет переход из состояния показа видео в состояние паузы и обратно. Сначала мы добавим это поведение в набор для всех состояний, что очень просто, поскольку оно тоже имеет в качестве параметра объект NetStream и нам не потребуется дополнительный импорт:

```
function doPause(ns:NetStream):void;
```

Теперь мы изменим файл *State.as*, добавив в него эту новую функцию, как показано в примере 10.14 в строке 9:

Пример 10.14. State.as

```
1 package
2 {
3     // Интерфейс состояния
4     import flash.net.NetStream;
5     interface State
6     {
```

```
        function startPlay(ns:NetStream,flv:String):void;
        function stopPlay(ns:NetStream):void;
        function doPause(ns:NetStream):void;
    }
}
```

Одним из правил использования интерфейсов является то, что все его имплементации должны включать в себя все указанные в нем методы. Следующим шагом мы добавим новый метод в состояния `Stop` и `Play`, как показано в примере 10.15:

Пример 10.15. StopState.as

```
package
{
    // Состояние «проигрывание остановлено»
    import flash.net.NetStream;

    class StopState implements State
    {
        private var videoWorks:VideoWorks;
        public function StopState(videoWorks:VideoWorks)
        {
            trace("--Stop State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            ns.play(flv);
            trace("Begin playing");
            videoWorks.setState(videoWorks.getPlayState());
        }
        public function stopPlay(ns:NetStream):void
        {
            trace("You're already stopped");
        }
        public function doPause(ns:NetStream):void
        {
            trace("Cannot go to Stop from Pause.");
        }
    }
}
```

Метод `doPause()` был добавлен в файл *StopState.as*, как показано в строке 24. Но нет никакого смысла в переходе в состояние паузы из состояния стоп, поскольку ничего не показывается, и нечего ставить на паузу. Таким образом, в этом методе присутствует только выражение `trace()` (в строке 26), которое говорит, что подобный переход невозможен.

Работа с состоянием Play происходит совершенно по-другому. По существу состояние Pause является его подмножеством. При работе с паузами можно представлять себе показ видео как «показ без паузы» и «показ с паузой». В ActionScript 3.0 для этого можно использовать новые методы NetStream.pause() («пауза») и NetStream.resume() («возобновить показ»). Но тогда нам потребовались бы два состояния вместо одного. Поэтому пока просто измените файл *PlayState.as*, как показано в примере 10.16 в строках 24–29.

Пример 10.16. PlayState.as

```
 1 package
 2 {
 3     // Состояние «проигрывание»
 4     import flash.net.NetStream;
 5
 6     class PlayState implements State
 7     {
 8         private var videoWorks:VideoWorks;
 9         public function PlayState(videoWorks:VideoWorks)
10         {
11             trace("--Play State--");
12             this.videoWorks=videoWorks;
13         }
14         public function startPlay(ns:NetStream,flv:String):void
15         {
16             trace("You're already playing");
17         }
18         public function stopPlay(ns:NetStream):void
19         {
20             ns.close();
21             trace("Stop playing.");
22             videoWorks.setState(videoWorks.getStopState());
23         }
24         public function doPause(ns:NetStream):void
25         {
26             ns.togglePause();
27             trace("Begin pause.");
28             videoWorks.setState(videoWorks.getPauseState());
29         }
30     }
31 }
```

Новый класс состояния «пауза» PauseState имплементирует интерфейс состояний State. Его функция doPause() использует тот же метод ns.togglePause(), что применяется в классе PlayState, как показано в примере 10.17. Однако он переходит не в состояние PauseState, а в PlayState. Интересной особенностью таких методов-переключателей является то, что совсем разные функции могут использовать один и тот же метод класса.

Пример 10.17. PauseState.as

```
package
{
    // Состояние «пауза»
    import flash.net.NetStream;

    class PauseState implements State
    {
        var videoWorks:VideoWorks;
        public function PauseState(videoWorks:VideoWorks)
        {
            trace("--Pause State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You have to go to unpause");
        }
        public function stopPlay(ns:NetStream):void
        {
            trace("Don't go to Stop from Pause");
        }
        public function doPause(ns:NetStream):void
        {
            ns.togglePause();
            trace("Quit pausing.");
            videoWorks.setState(videoWorks.getPlayState());
        }
    }
}
```

Как и в случае других классов состояний, вам необходимо добавить представителя `PauseState` в контекстный класс `VideoWorks`, как показано в примере 10.18. Помните, что этот класс содержит в себе все методы установщики и получатели, и вам обязательно будет нужно вызывать в них паузы в вашем приложении, что очень просто, поскольку требует всего лишь одной переменной для представителя `PauseState`, функции `doPause()` и получателя состояния `PauseState`.

Пример 10.18. VideoWorks.as

```
package
{
    // Контекстный класс
    import flash.net.NetStream;

    class VideoWorks
    {
        private var playState:State;
        private var stopState:State;
```

```
        private var pauseState:State;
        private var state:State;
        public function VideoWorks()
        {
            trace("Video Player is on");
            playState = new PlayState(this);
            stopState = new StopState(this);
            pauseState = new PauseState(this);
            state=stopState;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            state.startPlay(ns,flv);
        }
        public function stopPlay(ns:NetStream):void
        {
            state.stopPlay(ns);
        }
        public function doPause(ns:NetStream):void
        {
            state.doPause(ns);
        }
        public function setState(state:State):void
        {
            trace("A new state is set");
            this.state=state;
        }
        public function getState():State
        {
            return state;
        }
        public function getPlayState():State
        {
            return this.playState;
        }
        public function getStopState():State
        {
            return this.stopState;
        }
        public function getPauseState():State
        {
            return this.pauseState;
        }
    }
}
```

Чтобы протестировать новый класс паузы, все, что вам потребуется сделать в тестовом классе, – это добавить еще одну кнопку. Хотя три кнопки придется немного больше позиционировать на сцене, расширение шаблона проектирования Состояние получается очень простым.

Все изменения при добавлении дополнительной кнопки расположены в строчках 45–48 и 53. Так что просто отредактируйте файл *TestVid.as*, как показано в примере 10.19, и сохраните его как *TestPause.as*.

Пример 10.19. TestPause.as

```
 1 package
 2 {
 3     // Имплементация приложения FMS2 и тестирование автомата состояний
 4     import flash.display.Sprite;
 5     import flash.net.NetConnection;
 6     import flash.net.NetStream;
 7     import flash.media.Video;
 8     import flash.text.TextField;
 9     import flash.text.TextFieldType;
10     import flash.events.MouseEvent;
11     import flash.events.NetStatusEvent;
12
13     public class TestPause extends Sprite
14     {
15         private var nc:NetConnection=new NetConnection();
16         private var ns:NetStream;
17         private var vid:Video=new Video(320,240);
18         private var vidTest:VideoWorks;
19         private var playBtn:NetBtn;
20         private var stopBtn:NetBtn;
21         private var flv:String;
22         private var flv_txt:TextField;
23         private var dummy:Object;
24
25         public function TestPause ()
26         {
27             nc.connect (null);
28             ns=new NetStream(nc);
29             addChild (vid);
30             vid.x=stage.stageWidth / 2 - vid.width / 2;
31             vid.y=stage.stageHeight / 2 - vid.height / 2;
32
33             // Создаем автомат состояний
34             vidTest=new VideoWorks  ;
35
36             // Кнопки проигрывания, остановки и паузы
37             playBtn=new NetBtn("Play");
38             addChild (playBtn);
39             playBtn.x=stage.stageWidth / 2 - 100 + playBtn.width / 2;
40             playBtn.y=350;
41             stopBtn=new NetBtn("Stop");
42             addChild (stopBtn);
43             stopBtn.x=stage.stageWidth / 2 - stopBtn.width / 2;
44             stopBtn.y=350;
```

```
            var pauseBtn:NetBtn=new NetBtn("Pause");
            addChild (pauseBtn);
            pauseBtn.x=(stage.stageWidth / 2 + 100) - pauseBtn.width;
            pauseBtn.y=350;

            // Добавляем обработчики событий
            playBtn.addEventListener (MouseEvent.CLICK,doPlay);
            stopBtn.addEventListener (MouseEvent.CLICK,doStop);
            pauseBtn.addEventListener (MouseEvent.CLICK,pauseNow);

            // Добавляем текстовое поле
            flv_txt=new TextField  ;
            flv_txt.border=true;
            flv_txt.borderColor=0x9e0039;
            flv_txt.background=true;
            flv_txt.backgroundColor=0xfab383;
            flv_txt.type=TextFieldType.INPUT;
            flv_txt.x=stage.stageWidth / 2 - 45;
            flv_txt.y=10;
            flv_txt.width=90;
            flv_txt.height=16;
            addChild (flv_txt);

            // Для предотвращения ошибки MetaData
            dummy=new Object  ;
            ns.client=dummy;
            dummy.onMetaData=getMeta;

            // NetStream
            ns.addEventListener (NetStatusEvent.NET_STATUS,flvCheck);
        }
        // MetaData
        private function getMeta (mdata:Object):void
        {
            trace (mdata.duration);
        }
        // Работаем с flv
        private function flvCheck (event:NetStatusEvent):void
        {
            switch (event.info.code)
            {
                case "NetStream.Play.Stop" :
                    vidTest.stopPlay (ns);
                    vid.clear ();
                    break;
                case "NetStream.Play.StreamNotFound" :
                    vidTest.stopPlay (ns);
                    flv_txt.text="File not found";
                    break;
            }
        }
```

```
        // Начинаем проигрывание
        private function doPlay (e:MouseEvent):void
        {
            if (flv_txt.text != "" && flv_txt.text !=
                "Provide file name")
            {
                flv_txt.textColor=0x000000;
                flv=flv_txt.text + ".flv";
                vidTest.startPlay (ns,flv);
                vid.attachNetStream (ns);
            }
            else
            {
                flv_txt.textColor=0xcc0000;
                flv_txt.text="Provide file name";
            }
        }
        // Останавливаем проигрывание
        private function doStop (e:MouseEvent):void
        {
            vidTest.stopPlay (ns);
            vid.clear ();
        }
        // Ставим проигрывание на паузу
        function pauseNow (e:MouseEvent):void
        {
            vidTest.doPause (ns);
        }
    }
}
```

На рис. 10.6 показано, как должен выглядеть ваш видеоплеер.

Добавление новых состояний и возможности записи

Теперь, когда наша система поддерживает функции простого *flv*-плеера, нашей задачей будет добавление в нее двух новых дополнительных состояний и адаптация нашего конечного автомата до полноценного приложения Flash Media Server 2 (FMS2). Если говорить лишь в терминах шаблона проектирования, то мы добавим только два новых состояния – `Record` («запись») и `Append` («добавление»).

Переход от приложения Flash к приложению FMS2 потребует серьезных изменений в нашем *fla*-сценарии, необходимых для подключения к серверу и включения объектов видеокамеры и микрофона. Во всем остальном, как вы увидите, добавление двух новых состояний записи и добавления видео будет относительно простым.

Рис. 10.6. *Плеер с новой кнопкой Pause*

Настройка ActionScript 3.0 для FMS2

Как вы уже видели, добавление новых состояний – несложное дело. Однако если вы добавляете еще и FMS2, то для обеспечения корректной работы вашего приложения вам необходимо позаботиться о некоторых моментах. Поскольку ActionScript 3.0 и клиентский ActionScript (Client-Side ActionScript, CSAS), используемый в FMS2, немного отличаются, вам нужно импортировать класс `net.ObjectEncoding`. После этого ActionScript 3.0 и клиентский ActionScript смогут работать вместе. В ActionScript 3.0 по умолчанию форматом командных сообщений (Action Message Format (AMF)) является AMF3, а для FMS2 требуется AMF0. Поэтому вам нужно перенастроить класс `ObjectEncoding` на AMF0 с помощью строки

```
NetConnection.defaultObjectEncoding=flash.net.ObjectEncoding.AMF0;
```

Эта строка должна быть в вашей имплементации шаблона проектирования Состояние, а не в классах, которые его составляют (см. пример 10.27).

Вы увидите, что в классе состояния `StopState`, который мы сделаем для данного приложения (пример 10.21), мы будем совершенно по-другому

работать с методом `NetStream.play()`. При добавлении второго параметра данный метод превращается в метод пользовательского ActionScript. Но, поскольку мы импортируем класс `flash.net.ObjectEncoding`, эти две версии ActionScript могут работать у нас вместе.

Flash Media Server 2 – это использующий технологию open socket видеосервер, созданный компанией Adobe. Вы можете свободно загрузить его версию для разработчиков с сайта Adobe по адресу: *http://www.adobe.com/products/flashmediaserver/*.

Для его установки вам потребуется Windows или Linux-сервер или просто ваш компьютер с ОС Windows. На Макинтошах для его установки нужно будет перейти в режим Windows.

После инсталляции добавьте новый подкаталог *flvstate* в каталог *applications* FMS2. Кроме этого больше ничего не нужно делать с FMS2, разве что проверьте, что он работает, когда вы пользуетесь FMS приложением. (Для установки FMS2 следуйте указаниям в его документации.)

Адаптируемые состояния

Нашей первой задачей будет обновление схемы для нашего конечного автомата. Рисунок 10.7 показывает два новых состояния – `Append` (Добавить) и `Record` (Запись). Предыдущие три состояния остались практически без изменений. Обратите внимание, состояние `Stop` стало как бы центральным для всех переходов, кроме переключения режимов `Pause-Play`. Теперь, чтобы попасть в любое состояние, за исключением `Pause`, нужно пройти через состояние `Stop`.

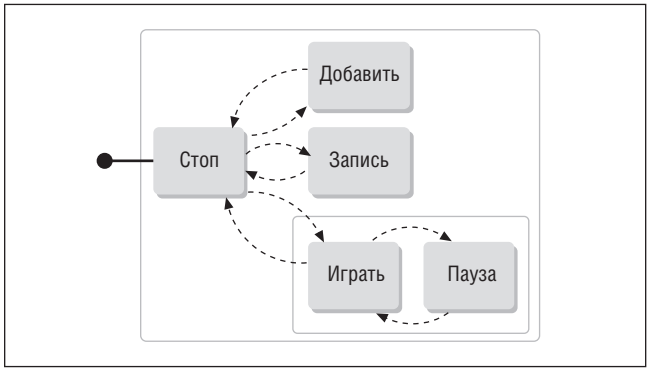


Рис. 10.7. *Диаграмма с пятью состояниями*

Как нами уже отмечалось в начале главы, диаграммы состояний позволяют легко увидеть все необходимые изменения в программе. При добавлении в нее еще двух состояний, `Append` (добавить) и `Record` (запись), все остальные состояния и контекстный класс тоже должны изменить-

ся. Однако нам не придется при этом менять громадное число условных выражений. В тестировании приложения также произойдут изменения, но поскольку они касаются больше вопросов использования нашего конечного автомата, чем его самого, они будут рассматриваться нами отдельно. В следующих сценариях из примеров с 10.20 по 10.26 добавлены необходимые изменения в работу нашего конечного автомата.

Пример 10.20. State.as

```
1 package
2 {
3     // Интерфейс состояния
4     import flash.net.NetStream;
5     interface State
6     {
7         function startPlay(ns:NetStream,  flv:String):void;
8         function startRecord(ns:NetStream, flv:String):void;
9         function startAppend(ns:NetStream, flv:String):void;
10        function stopAll(ns:NetStream):void;
11        function doPause(ns:NetStream):void;
12    }
13 }
```

Обратите внимание, что имя метода остановки показа видео было изменено на `stopAll()` (остановить все) в строке 10. Это сделано потому, что в различных состояниях остановка имеет разный смысл. Она может означать остановку записи видео или его добавления помимо остановки его показа. Таким образом, данное изменение показывает, что теперь мы выполняем не одно действие. И здесь мы опять пользуемся возможностями полиморфизма.

В классе `StopState` имеется один метод из `NetStream`, являющийся частью более старой версии ActionScript, или клиентского ActionScript из FMS2. В ActionScript 3.0 метод `NetStream` ожидает только один аргумент – строку с адресом URL *flv*-файла. Однако в клиентском ActionScript вы можете добавить ему и второй параметр, который будет специфицировать, какой тип потокового видео показывать. Чтобы ActionScript 3.0 мог работать с классами из предыдущих версий ActionScript, использующих потоковые объекты, в приложении должен быть импортирован класс `ObjectEncoding`. Если это сделано, ActionScript 3.0 может полностью интегрировать все эти объекты и их параметры. В строке 20 показан второй аргумент для метода `NetStream.play()`.

Пример 10.21. StopState.as

```
1 package
2 {
3     // Состояние «проигрывание остановлено»
4     import flash.net.NetStream;
5
```

```
    class StopState implements State
    {
        var videoWorks:VideoWorks;
        public function StopState(videoWorks:VideoWorks)
        {
            trace("--Stop State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            // Заметка: второй параметр (0) – указывает на FLV-файл
            // метод NetStream работает с ActionScript
            // клиентской части, но в ActionScript 3.0 он тоже работает,
            // так как импортирован класс ObjectEncoding.
            ns.play(flv,0);
            trace("Begin playing");
            videoWorks.setState(videoWorks.getPlayState());
        }
        public function startRecord(ns:NetStream,flv:String):void
        {
            ns.publish(flv,"record");
            trace("Begin recording");
            videoWorks.setState(videoWorks.getRecordState());
        }
        public function startAppend(ns:NetStream,flv:String):void
        {
            ns.publish(flv,"append");
            trace("Begin appending");
            videoWorks.setState(videoWorks.getAppendState());
        }
        public function stopAll(ns:NetStream):void
        {
            trace("You're already stopped");
        }
        public function doPause(ns:NetStream):void
        {
            trace("Must be playing to pause.");
        }
    }
}
```

Может показаться несколько ироничным, что состояние с имплементацией самых используемых в видео методов называется `Stop`. Но только из этого состояния возможны переходы к состояниям `Record` и `Append`. Класс `PlayState` показан в примере 10.22, из него могут совершаться переходы и в состояние `Stop`, и в состояние `Pause`.

Пример 10.22. PlayState.as

```
package
{
```

```
    // Состояние «проигрывание»
    import flash.net.NetStream;

    class PlayState implements State
    {
        var videoWorks:VideoWorks;
        public function PlayState(videoWorks:VideoWorks)
        {
            trace("--Play State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You're already playing");
        }
        public function stopAll(ns:NetStream):void
        {
            ns.close();
            trace("Stop playing.");
            videoWorks.setState(videoWorks.getStopState());
        }
        public function startRecord(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function startAppend(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function doPause(ns:NetStream):void
        {
            ns.togglePause();
            trace("Start pausing.");
            videoWorks.setState(videoWorks.getPauseState());
        }
    }
}
```

Состояние `Play` немного изменилось со своей предыдущей версии. При показе видео структура его переходов не позволяет напрямую перейти к записи или добавлению видео. Если бы это было возможно, то существовал бы риск перезаписи того видео, которое вы смотрели, новым записываемым видео. Поэтому из состояния `Play` возможны только переходы в состояния `Stop` и `Pause`. Класс состояния `Pause` показан в примере 10.23.

Пример 10.23. PauseSate.as

```
package
{
    // Состояние «пауза»
```

```
4     import flash.net.NetStream;
5
6     class PauseState implements State
7     {
8         var videoWorks:VideoWorks;
9         public function PauseState(videoWorks:VideoWorks)
10        {
11            trace("--Pause State--");
12            this.videoWorks=videoWorks;
13        }
14        public function startPlay(ns:NetStream,flv:String):void
15        {
16            trace("You have to go to unpause");
17        }
18        public function stopAll(ns:NetStream):void
19        {
20            trace("Don't go to Stop from Pause");
21        }
22        public function startRecord(ns:NetStream,flv:String):void
23        {
24            trace("You have to stop first.");
25        }
26        public function startAppend(ns:NetStream,flv:String):void
27        {
28            trace("You have to stop first.");
29        }
30        public function doPause(ns:NetStream):void
31        {
32            ns.togglePause();
33            trace("Quit pausing.");
34            videoWorks.setState(videoWorks.getPlayState());
35        }
36    }
37
38 }
```

Как и в классе состояния `Play`, в классе состояния `Pause`, показанном в примере 10.23, произошло немного изменений, поскольку оно только переключается между показом видео и его паузой и не делает пауз при записи нового видео.

Класс `RecordState` описывает новое для нас состояние. Но если вы посмотрите на него внимательно, то увидите, что он довольно похож на класс состояния `PlayState`, за исключением того, что он не может совершать переходы в состояние паузы. Вспомним, что, находясь в состоянии записи, мы можем ее только остановить, и именно потому метод `stopAll()` имплементирован в классе этого состояния, показанном в примере 10.24. Сохраните данный новый класс под именем заголовка его примера в том же каталоге, что и остальные файлы приложения.

Пример 10.24. RecordState.as

```
package
{
    // Состояние «запись»
    import flash.net.NetStream;

    class RecordState implements State
    {
        var videoWorks:VideoWorks;
        public function RecordState(videoWorks:VideoWorks)
        {
            trace("--Record State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function stopAll(ns:NetStream):void
        {
            ns.close();
            trace("Stop recording.");
            videoWorks.setState(videoWorks.getStopState());
        }
        public function startRecord(ns:NetStream,flv:String):void
        {
            trace("You're already recording");
        }
        public function startAppend(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function doPause(ns:NetStream):void
        {
            trace("Must be playing to pause.");
        }
    }
}
```

Класс `AppendState`, показанный в примере 10.25, является еще одним новым классом состояния. Внешне он почти идентичен классу `RecordState`. Однако это не должно быть для нас большим сюрпризом, поскольку процессы записи и добавления видео очень похожи. Сохраните данный класс в том же каталоге, что и другие файлы для этого приложения, взяв заголовок примера в качестве его имени.

Пример 10.25. AppendState.as

```
package
{
```

```
    // Состояние «добавление»
    import flash.net.NetStream;


    class AppendState implements State
    {
        var videoWorks:VideoWorks;
        public function AppendState(videoWorks:VideoWorks)
        {
            trace("--Append State--");
            this.videoWorks=videoWorks;
        }
        public function startPlay(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function stopAll(ns:NetStream):void
        {
            ns.close();
            trace("Stop appending.");
            videoWorks.setState(videoWorks.getStopState());
        }
        public function startRecord(ns:NetStream,flv:String):void
        {
            trace("You have to stop first.");
        }
        public function startAppend(ns:NetStream,flv:String):void
        {
            trace("You're already appending");
        }
        public function doPause(ns:NetStream):void
        {
            trace("Must be playing to pause.");
        }
    }
}
```

В новой версии класса `VideoWorks`, показанном в примере 10.26, не произошло крупных изменений, кроме появления представителей классов новых состояний. Такие малые изменения как раз и показывают нам всю силу структуры шаблона проектирования Состояние. А ведь в приложение было не только добавлено два новых состояния для записи и добавления видео, но и само оно было переделано с простого Flash плеера с последовательной загрузкой в приложение работы с потоковым видео FMS2.

Пример 10.26. VideoWorks.as

```
package
{
    // Контекстный класс
```

```
4     import flash.net.NetStream;
5     public class VideoWorks
6     {
7         var playState:State;
8         var stopState:State;
9         var recordState:State;
10        var appendState:State;
11        var pauseState:State;
12        var state:State;
13        public function VideoWorks ()
14        {
15            trace ("Video Player is on");
16            playState = new PlayState(this);
17            stopState = new StopState(this);
18            recordState = new RecordState(this);
19            appendState = new AppendState(this);
20            pauseState=new PauseState(this);
21            state=stopState;
22        }
23        public function startPlay (ns:NetStream,flv:String):void
24        {
25            state.startPlay (ns,flv);
26        }
27        public function startRecord (ns:NetStream,flv:String):void
28        {
29            state.startRecord (ns,flv);
30        }
31        public function startAppend (ns:NetStream,flv:String):void
32        {
33            state.startAppend (ns,flv);
34        }
35        public function stopAll (ns:NetStream):void
36        {
37            state.stopAll (ns);
38        }
39        public function doPause (ns:NetStream):void
40        {
41            state.doPause (ns);
42        }
43        public function setState (state:State):void
44        {
45            trace ("A new state is set");
46            this.state=state;
47        }
48        public function getState ():State
49        {
50            return state;
51        }
52        public function getPlayState ():State
53        {
54            return this.playState;
```

```
        }
        public function getRecordState ():State
        {
            return this.recordState;
        }
        public function getAppendState ():State
        {
            return this.appendState;
        }
        public function getPauseState ():State
        {
            return this.pauseState;
        }
        public function getStopState ():State
        {
            return this.stopState;
        }
    }
}
```

Наш тестовый класс модуля *TestFMS.as* из примера 10.27 теперь стал немного больше, чем наши прошлые тестовые классы. (На самом деле его стоило бы разбить на подклассы, но нашей главной задачей является демонстрация шаблона проектирования Состояние, потому мы сохранили простой тестовый класс.) Обязательно поместите копии файлов *NetBtn.as* и *BtnState.as* в каталоге с остальными файлами вашего приложения.

Пример 10.27. TestFMS.as

```
package
{
    // Тестирующий модуль
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.net.ObjectEncoding;
    import flash.media.Video;
    import flash.media.Camera;
    import flash.media.Microphone;
    import flash.text.TextField;
    import flash.text.TextFieldType;
    import flash.events.MouseEvent;
    import flash.events.NetStatusEvent;

    public class TestVidFMS extends Sprite
    {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var dummy:Object;
        private var flv_txt:TextField;
```

```
22          private var cam:Camera;
23          private var mic:Microphone;
24          private var stateVid:VideoWorks;
25          private var playCheck:Boolean;
26          private var pauseCheck:Boolean;
27          private var playBtn:NetBtn;
28          private var stopBtn:NetBtn;
29          private var pauseBtn:NetBtn;
30          private var recordBtn:NetBtn;
31          private var appendBtn:NetBtn;
32
33          public function TestVidFMS ()
34          {
35              //************
36              // Добавляем текстовое поле
37              //************
38              flv_txt= new TextField();
39              flv_txt.border=true;
40              flv_txt.background=true;
41              flv_txt.backgroundColor=0xfab383;
42              flv_txt.type=TextFieldType.INPUT;
43              flv_txt.x=(550/2)-45;
44              flv_txt.y=15;
45              flv_txt.width=90;
46              flv_txt.height=18;
47              addChild (flv_txt);
48              // автомат состояний FMS
49              NetConnection.defaultObjectEncoding =
                                              flash.net.ObjectEncoding.AMF0;
50              nc = new NetConnection();
51              nc.objectEncoding = flash.net.ObjectEncoding.AMF0;
52              nc.addEventListener (NetStatusEvent.NET_
                                              STATUS,checkHookupStatus);
53              // используем собственный домен/IP адрес для RTMP
54              nc.connect ("rtmp://192.168.0.11/flvstate/flv");
55              // ИЛИ создаем локальное соединение
56              //nc.connect("rtmp:/flvstate/flv");
57              //nc.connect(null);
58
59              // Установки камеры и микрофона
60              cam = Camera.getCamera();
61              cam.setMode (320,240,15);
62              cam.setKeyFrameInterval (30);
63              cam.setQuality (0,80);
64              mic = Microphone.getMicrophone();
65              mic.rate=11;
66
67              // Добавляем видеообъект
68              vid=new Video(320,240);
69
70              addChild (vid);
```

```
71              vid.x=(550/2)-(320/2);
72              vid.y=40;
73              setLocal ();
74
75              // Реализуем автомат состояний
76              stateVid=new VideoWorks;
77
78              // Кнопки проигрывания, остановки, паузы, записи и добавления
79              playBtn=new NetBtn("Play");
80              addChild (playBtn);
81              playBtn.x=(550/2)-(320/2);
82              playBtn.y=300;
83              var playCheck:Boolean=false;
84
85              recordBtn=new NetBtn("Record");
86              addChild (recordBtn);
87              recordBtn.x=(550/2)+((320/2)-60);
88              recordBtn.y=300;
89
90              appendBtn=new NetBtn("Append");
91              addChild (appendBtn);
92              appendBtn.x=(550/2)+((320/2)-60);
93              appendBtn.y=330;
94
95              stopBtn=new NetBtn("Stop");
96              addChild (stopBtn);
97              stopBtn.x=(550/2)-25;
98              stopBtn.y=300;
99
100             pauseBtn=new NetBtn("Pause");
101             addChild (pauseBtn);
102             pauseBtn.x=(550/2)-(320/2);
103             pauseBtn.y=330;
104             pauseCheck=true;
105
106             // Добавляем обработчики событий
107             playBtn.addEventListener (MouseEvent.CLICK,doPlay);
108             stopBtn.addEventListener (MouseEvent.CLICK,doStop);
109             recordBtn.addEventListener (MouseEvent.CLICK,doRecord);
110             appendBtn.addEventListener (MouseEvent.CLICK,doAppend);
111             pauseBtn.addEventListener (MouseEvent.CLICK,doPause);
112
113         }
114         // Добавляем функции управления
115         function setNet ()
116         {
117             vid.attachNetStream (ns);
118         }
119         function setLocal ()
120         {
```

```
121             vid.attachCamera (cam);
122         }
123         var flv:String;
124         function doPlay (e:MouseEvent):void
125         {
126             if (flv_txt.text != “” && flv_txt.text != “Provide file name”)
127             {
128                 setNet ();
129                 flv_txt.textColor=0x000000;
130                 flv=flv_txt.text;
131                 stateVid.startPlay (ns,flv);
132                 if (! playCheck)
133                 {
134                     playCheck=true;
135                 }
136             }
137             else
138             {
139                 flv_txt.textColor=0xcc0000;
140                 flv_txt.text=”Provide file name”;
141             }
142         }
143         function doRecord (e:MouseEvent):void
144         {
145             if (flv_txt.text != “” && flv_txt.text != “Provide file name”)
146 
147             {
148                 ns.attachAudio (mic);
149                 ns.attachCamera (cam);
150                 flv_txt.textColor=0x000000;
151                 flv=flv_txt.text;
152                 stateVid.startRecord (ns,flv);
153                 if (! playCheck)
154                 {
155                     playCheck=true;
156                 }
157             }
158             else
159             {
160                 flv_txt.textColor=0xcc0000;
161                 flv_txt.text=”Provide file name”;
162             }
163         }
164         function doAppend (e:MouseEvent):void
165         {
166             if (flv_txt.text != “” && flv_txt.text != “Provide file name”)
167             {
168                 ns.attachAudio (mic);
169                 ns.attachCamera (cam);
170                 flv_txt.textColor=0x000000;
```

```
171                 flv=flv_txt.text;
172                 stateVid.startAppend (ns,flv);
173                 if (! playCheck)
174                 {
175                     playCheck=true;
176                 }
177             }
178             else
179             {
180                 flv_txt.textColor=0xcc0000;
181                 flv_txt.text="Provide file name";
182             }
183         }
184         function doPause (e:MouseEvent):void
185         {
186             if (pauseCheck)
187             {
188                 pauseCheck=false;
189                 if (playCheck)
190                 {
191                     stopBtn.visible=false;
192                 }
193                 stateVid.doPause (ns);
194             }
195             else
196             {
197                 pauseCheck=true;
198                 stopBtn.visible=true;
199                 stateVid.doPause (ns);
200 
201             }
202         }
203         function doStop (e:MouseEvent):void
204         {
205             playCheck=false;
206             stateVid.stopAll (ns);
207             vid.clear ();
208             setLocal ();
209         }
210         // Проверяем соединение, создаем поток
211         // и подключаем обработчик события MetaData
212         function checkHookupStatus (event:NetStatusEvent):void
213         {
214             if (event.info.code == "NetConnection.Connect.Success")
215             {
216                 ns = new NetStream(nc);
217                 dummy=new Object();
218                 ns.client=dummy;
219                 dummy.onMetaData=getMeta;
220                 ns.addEventListener (NetStatusEvent.NET_STATUS,flvCheck);
```

```
221             }
222         }
223         // MetaData
224         function getMeta (mdata:Object):void
225         {
226             trace (mdata.duration);
227         }
228         // Работаем с flv
229         private function flvCheck (event:NetStatusEvent):void
230         {
231             switch (event.info.code)
232             {
233                 case "NetStream.Play.Stop" :
234                     stateVid.stopAll(ns);
235                     setLocal();
236                     break;
237                 case "NetStream.Play.StreamNotFound" :
238                     stateVid.stopAll(ns);
239                     flv_txt.text="File not found";
240                     setLocal();
241                     break;
242             }
243         }
244     }
245 }
```

При тестировании данного приложения очень важно убедиться, что у вас правильно сконфигурирован протокол RTMP (Real-Time Messaging Protocol). Если вы тестируете это приложение, используя ваш собственный компьютер в качестве платформы для FMS2, то вам нужно закомментировать строчку 54 и удалить символы комментария из строки 56. Это позволит вашему приложению работать на том же сервере, где находится ваш *swf*-файл. В противном случае отредактируйте строку 38, чтобы она правильно на него указывала.

Рисунок 10.8 показывает, как примерно должно выглядеть ваше приложение при его правильной работе. Как вы видите, оно не сильно изменилось, поскольку в него добавились всего две новые кнопки. Однако его функциональность стала значительно разнообразнее с появлением возможностей записи и добавления видео в файлы.

Заключение

Для нас привычно мыслить парами существительное–глагол, поэтому требуется несколько приноровиться к шаблонам, в которых используется глагольный элемент пары. О состояниях можно думать как о состояниях действия, таких как показ видео и запись, а также как о состояниях покоя и пассивности. А переходы между состояниями описываются с помощью глаголоцентричных их характеристик.

Рис. 10.8. *Запись видео*

В центре внимания шаблона проектирования Состояние всегда находятся действия, а не характеристики объектов. Прилагаемые к ним диаграммы состояний позволяют гораздо проще и эффективнее проектировать приложения. Этот процесс начинается с того, что обычно оставляют на конец разработки, – с установления всех связей между различными элементами. При наличии готовых диаграмм состояний перевод их в шаблон проектирования Состояние относительно несложен, поскольку его структура классов весьма проста и наглядна. Каждое состояние имеет свой собственный класс, и классы состоят из различных переходов между состояниями, определенных в интерфейсе состояния. Их общему контекстному классу остается только собрать всех их вместе. Поэтому, если ваш проект включает в себя различные действия, которые должны быть организованы и скоординированы, подумайте об использовании в нем конечных автоматов с помощью шаблона проектирования Состояние.

11

Шаблон Стратегия

Применяй стратегию широко, корректно и открыто. Тогда ты начнешь думать о вещах в широкой перспективе и, приняв пустоту как Путь, увидишь Путь как пустоту.

Миямото Мусаси (16–17 вв.; японский разработчик шаблона проектирования Метод Меча (Кэндзюцу), который напоминает нам, что в ActionScript 3.0 слово «void» (пустота) всегда пишется с маленькой буквы.)

Все воины знают форму сил, благодаря которой мы добиваемся победы, но никто не знает той формы сил, с помощью которой мы управляем победой.

Сунь Цзы

Как бы ни была красива стратегия, время от времени нужно смотреть и на результаты.

Уинстон Черчилль (Замечание, сделанное после реализации программы «Hello World» с помощью шаблона проектирования Стратегия.)

Что такое шаблон Стратегия?

Шаблон проектирования Стратегия иногда используют для иллюстрации принципов хорошего применения ООП, а не как шаблон проектирования, предназначенный для создания приложений. В одной из статей Эджербо (Agerbo) и Корнилс (Cornils) предложили несколько правил и характеристик, позволяющих судить об имеющихся шаблонах проектирования. Одно из таких правил постулировало, что *шаблон*

проектирования не должен быть непосредственно связан с объектно-ориентированным способом мышления. После чего авторы перешли к отрицанию шаблона проектирования Стратегия, поскольку он является не чем иным, как набором фундаментальных принципов ООП!

Но чем бы ни был шаблон Стратегия, истинным шаблоном проектирования или способом изучения объектно-ориентированного программирования, он многое может предложить вам. Поскольку его модель классов практически идентична модели классов шаблона проектирования Состояние, нам придется потратить некоторое время на то, чтобы понять уникальность каждого из этих шаблонов и определить для них области приложений.

Ключевые характеристики

При изучении ключевых характеристик шаблона Стратегия полезно сравнить их с такими же характеристиками шаблона Состояние из предыдущей главы 10. Такое сравнение поможет вам понять не только разницу между этими шаблонами, но и каждый из них сам по себе.

Следующие ключевые характеристики используются для описания шаблона Стратегия, но вы найдете их практически в любых рекомендациях по правильному использованию объектно-ориентированного программирования:

- Определяет и инкапсулирует набор алгоритмов
- Делает инкапсулированные алгоритмы взаимозаменяемыми
- Позволяет алгоритмам меняться независимо от клиента, который их использует

Прежде чем вернуться к главе 10 и сравнить этот список с ключевыми характеристиками шаблона Состояние, взгляните на следующий раздел «Модель шаблона Стратегия». Потом делайте сравнение. Вы сразу же обнаружите, что ключевые характеристики шаблонов Состояние и Стратегия различаются, но, сравнивая также их модели, вы увидите все различия более ясно.

Поскольку характеристики шаблона Стратегия настолько фундаментальны для правильного применения ООП, мы отложим их более детальное обсуждение до раздела «Основные концепции ООП, используемые в шаблоне Стратегия». Как вы увидите, все эти характеристики применимы и к более общим принципам программирования на объектно-ориентированных языках, таких как ActionScript 3.0.

Модель шаблона Стратегия

Когда вы посмотрите на диаграмму классов шаблона проектирования Стратегия, у вас может возникнуть ощущение дежавю. На рис. 10.3 в главе 10 названия выглядят по-другому, но сам шаблон Состояние

очень похож. Не волнуйтесь. Они очень разные по предназначению и имплементации. Поэтому, смотря на рис. 11.1, помните о ключевых характеристиках шаблона проектирования Стратегия.

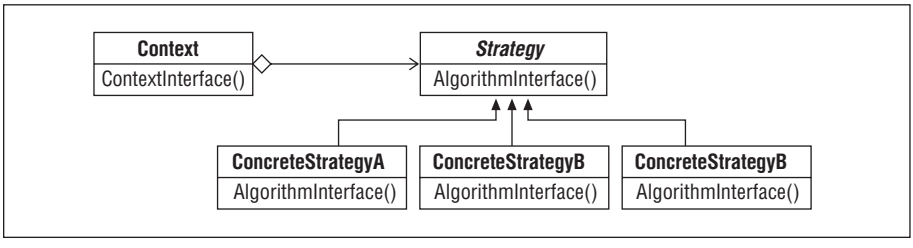


Рис. 11.1. *Диаграмма классов шаблона проектирования Стратегия*

Посмотрев на контекстный класс на диаграмме классов шаблона Стратегия, вы можете увидеть, что интерфейс стратегии является агрегированным классом контекста. Из-за такой агрегации мы можем сказать, что класс контекста (агрегатор) *владеет* интерфейсом стратегии (агрегируемый), точно так же как в шаблоне проектирования Состояние (см. рис. 10.3 в главе 10). Как вы увидите в примерах, контекстный класс агрегирует операции шаблона Стратегия. Поскольку агрегация является основой для многих шаблонов проектирования и важной частью ООП, вам стоит изучить ее реальное применение на примерах. Шаблон проектирования Стратегия – один из лучших источников подобных примеров использования агрегации на практике.

Ключевые концепции ООП, используемые в шаблоне Стратегия

Дэвид Гири (David Geary) в статье в JavaWorld.com от 26.04.02 отметил, что шаблон Стратегия включает в себя сразу несколько принципов ООП, выдвинутых Гаммой, Хелмом, Джонсоном и Влиссидесом. Один из этих принципов таков:

> Инкапсулируйте все, что меняется.

Эрик и Элизабет Фриман (Eric and Elisabeth Freeman) в книге «Head First Design Patterns» (O'Reilly, 2004 г.) тоже обращали на это внимание при обсуждении шаблона Стратегия. И мы не собираемся ломать эту традицию и выпускать из рассмотрения столь важную связь между изучаемым шаблоном и данной концепцией ООП.

Инкапсуляция вариаций

В главе 10 в шаблоне проектирования Состояние вы видели, что состояния, поведение которых варьировалось, например, в видеоплеере, были инкапсулированы в объекты состояний вместе со всем поведением, которое от них зависело (таким как показ видео и остановка операций).

В шаблоне Стратегия объекты стратегии инкапсулируют в себе алгоритмы. Общим для этих шаблонов является то, что они инкапсулируют то, что меняется. (В имени класса содержится намек на то, что в нем инкапсулировано. И в названии для класса стратегии, и в названии для класса состояния есть имя объекта, который они инкапсулируют в себе; проверьте диаграммы классов.)

Важно понять, почему «банда четырех» предложила инкапсулировать то, что изменяется. Если вы инкапсулируете те части, которые подвержены переменам, то при очередном изменении вашего приложения у вас будет меньше неприятных сюрпризов. Концепция инкапсуляции позволяет объектам в ООП вести себя подобно законченным объектам реального мира. Вместо того чтобы быть «приклеенными» и зависимыми от других элементов в программе, инкапсулированные объекты могут легко работать совместно с любыми новыми элементами, добавляемыми в это приложение, поскольку они могут изменяться независимо от других объектов.

Чтобы лучше понять, что значит инкапсулировать все изменяющееся, давайте рассмотрим скрипт из примера данной главы с и без такой инкапсуляции. В нем мы столкнемся с клоунами и с тем, что они делают на потеху детям.

Версия без инкапсуляции (псевдокод)

```
class Clown {
    function juggle() {
    // код жонглирования
    }
    function balloonAnimals() {
    // код фигурок из воздушных шаров
    }
}

class Bojo extends Clown {
    // уникальный Bojo
}

class Koka extends Clown {
    // уникальный Koka
}
```

Здесь алгоритмы по жонглированию и скручиванию зверушек из длинных шариков помещены в класс `Clown`. Они передаются конкретным его подклассам через наследование. Все это прекрасно работает, пока все клоуны могут выполнять данные трюки. Но что произойдет, если кто-то из клоунов не умеет жонглировать или делать зверушек из шариков? Или если потребуется добавить новое действие. Что нам тогда делать?

Так что давайте теперь инкапсулируем все алгоритмы в объект стратегии.

Инкапсулированные алгоритмы (псевдокод)

```
interface ClownTricks {
    function someTrick(): void;
}
class Juggle implements ClownTricks {
    function someTrick(): void {
     // алгоритм жонглирования
     }
}
class BalloonAnimals implements ClownTricks {
    function someTrick(): void {
     // алгоритм создания фигурок из воздушных шаров
     }
}
```

Теперь, используя класс `Clown` как агрегатор, программа может делегировать запросы на выполнение различных алгоритмов в конкретные подклассы этого класса. Если появятся новые трюки, нужно будет просто имплементировать их согласно интерфейсу `ClownTricks` и применять их при делегировании действий в конкретные подклассы класса `Clown`, которые будут его использовать. (Приложение, использующее все эти концепции, начинается с примера 11.6 данной главы.)

Инкапсулированные алгоритмы не были бы так полезны, если бы некоторые объекты не могли ими воспользоваться. Вот здесь и появляется делегирование, как вы увидите в следующем разделе.

Использование делегирования и делегатов

Шаблон проектирования Стратегия является прекрасным примером использования делегирования, при котором в выполнение одного запроса вовлекаются два объекта. Один из таких объектов получает запрос и затем делегирует выполнение действий *делегату*. Именно это и происходит между классами контекста и классами стратегий. Контекстные классы состоят из основного класса и его подклассов. Контекстные классы делегируют выполнение действий классам стратегии, состоящим из интерфейсов стратегий и имплементаций этих интерфейсов. Таким образом, контекстные классы делегируют полномочия классам стратегий.

Такой процесс делегирования начинается в главном контекстном классе. Он включает в себя переменные со ссылками на интерфейсы стратегии. Методы в контекстных классах производят окончательное делегирование полномочий. Конкретные контекстные классы наследуют переменные со ссылками из главного контекстного класса и используют их для спецификации, какое именно поведение им нужно от конкретных стратегий. Рисунок 11.2 дает визуальное описание процесса делегирования.

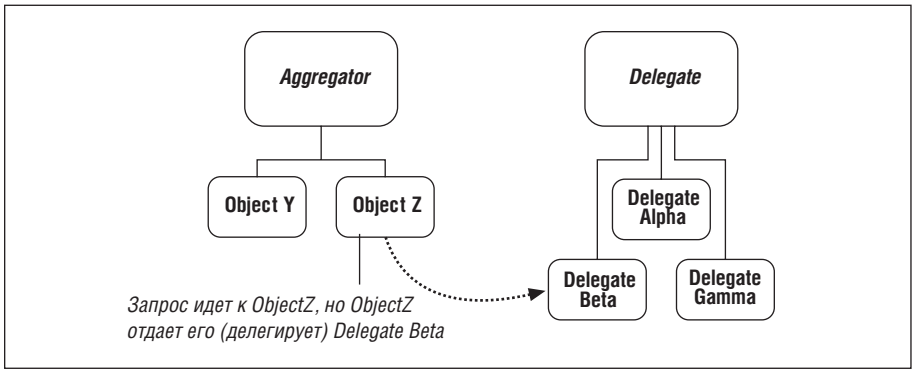


Рис. 11.2. *Процесс делегирования*

При вызове объекта класса, в котором есть ссылка на вызываемого делегата, этот делегат вызывает метод, содержащий требуемый инкапсулированный алгоритм. То есть происходит не наследование операций, а, можно сказать, в используемом объекте *имеется* метод, который обрабатывает запрос.

Минималистский абстрактный шаблон Стратегия

Этот первый пример будет воплощением самых основ шаблона проектирования Стратегия, он наглядно продемонстрирует вам его базовую структуру. Каждая его часть будет нами проанализирована в терминах того, что она делает для всего шаблона. Данный пример должен помочь вам ясно понять всю его конструкцию.

Чтобы как следует рассмотреть шаблон проектирования Стратегия, вам необходимо увидеть его в контексте, где несколько различных объектов делегируют выполнение действий различным делегатам. (Второй пример предоставит вам еще больше возможностей для этого.) Данный минималистский пример вам нужно рассматривать с точки зрения делегирования и хода его работы. На рис. 11.3, который будет приведен после всех листингов кода данного примера, вы сможете увидеть, как в нем выглядит вся паутина его связей. Но не волнуйтесь, эта иллюстрация просто показывает, как работает делегирование между различными объектами в приложении.

Использование делегирования вместе с контекстом

Контекстный класс с простым названием `Context`, показанный в примере 11.1, прежде всего является агрегатором. То есть он содержит в себе объекты стратегий. Поэтому в любом классе контекста, который вы найдете в шаблоне Стратегия, вы увидите ссылку на класс стратегии.

В данном примере это переменная с именем интерфейса стратегии с маленькой буквы. (Наше первое приложение будет состоять из примеров с 11.1 по 11.5.)

Пример 11.1. Context.as

```
package
{
    class Context
    {
        protected var strategy:Strategy;

        public function doStrategy():void
        {
            strategy.think();
        }
    }
}
```

Класс `Context` очень простой. Одна из причин этого заключается в том, что все необходимые ему методы делегируются имплементациям одного интерфейса стратегии `Strategy`. Давайте теперь посмотрим на такой интерфейс.

Добавление стратегии

В реализациях шаблона проектирования Стратегия вы будете, вероятно, видеть более одного интерфейса, в зависимости от числа разных типов необходимых в нем методов. Пример 11.2 показывает только один метод в интерфейсе.

Пример 11.2. Strategy.as

```
package
{
    interface Strategy
    {
        function think():void;
    }
}
```

Это будет наш первый шаг по инкапсуляции поведения объектов. Последующие имплементации интерфейса стратегии добавят в него необходимые детали.

Детали стратегии

В примере 11.3 класс конкретной стратегии добавляет метод `think()` («думать»), который что-то делает. В нашем случае он просто печатает сообщение.

Пример 11.3. ConcreteStrategy.as

```
package
{
    class ConcreteStrategy implements Strategy
    {
        public function think():void
        {
            trace("Great thoughts now...");
        }
    }
}
```

Больше делегирования в конкретном контексте

В конкретном классе контекста из примера 11.4 вы можете видеть, что работа делегируется конкретному классу стратегии.

Пример 11.4. ConcreteContext.as

```
package
{
    class ConcreteContext extends Context
    {
        public function ConcreteContext()
        {
            strategy = new ConcreteStrategy();
        }
    }
}
```

Собираем все части вместе

Наконец, чтобы протестировать нашу реализацию шаблона проектирования Стратегия, скрипт в примере 11.5 создает представителя конкретного класса контекста, но объявляет его через тип данных `Context`, что является примером использования в программировании интерфейсов, а не их имплементаций.

Пример 11.5. TestStrategy.as

```
package
{
    import flash.display.Sprite;

    public class TestStrategy extends Sprite
    {
        public function TestStrategy()
        {
            var thinker:Context= new ConcreteContext();
            thinker.doStrategy();
        }
```

```
        }
    }
```

Теперь, когда у нас есть реализация немного абстрактного шаблона Стратегия, в котором вы можете увидеть устройство различных его частей, иллюстрация, демонстрирующая все связи, делегирования и взаимодействия за раз, даст вам общее представление о процессе его работы. Рисунок 11.3 показывает все связи между различными частями нашего шаблона на одной картинке.

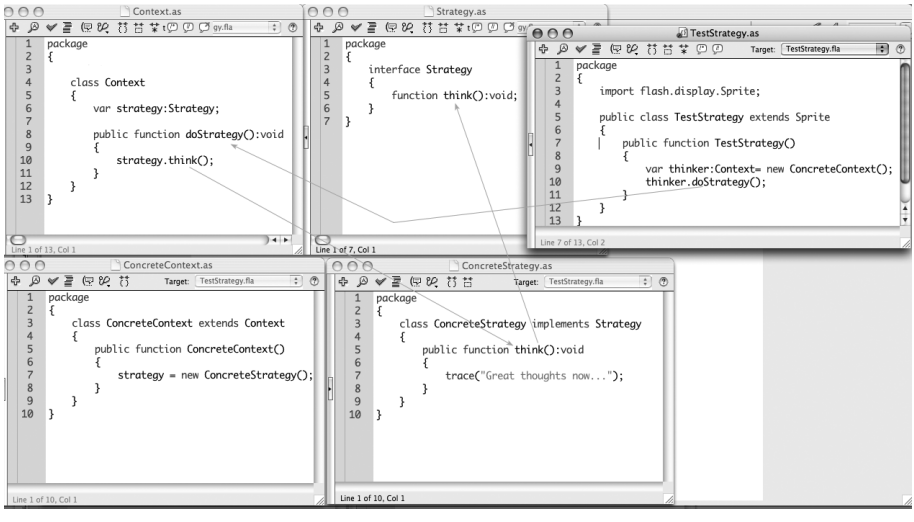


Рис. 11.3. *Взаимодействия делегирования*

На рис. 11.3 обратите внимание на то, что вызов

```
thinker.doStrategy();
```

является делегированием. Объект `thinker` в классе `TestStrategy` объявлен через тип `Context` и создан как объект `ConcreteContext`. Метод `doStrategy()` создается в классе `Context`, где мы и будем искать его в первую очередь, как показывает стрелочка от `doStrategy()` в классе `TestStrategy` к `doStrategy()` в классе `Context`.

Однако в классе `Context` метод `doStrategy()` делегирует действия в метод `think()`, поэтому нам надо найти его. Детали работы метода `think()` можно обнаружить в классе `ConcreteStrategy`. Поскольку объект класса `ConcreteStrategy` есть в объекте `thinker`, на самом деле он выполняет всю работу. Но поскольку класс `ConcreteContext` является подклассом `Context`, это делегирование структурировано именно в нем.

Наконец, мы можем проследить процесс делегирования обратно к интерфейсу стратегии `Strategy`. Поскольку метод `think()` объявлен именно в нем, мы понимаем, как он оказался делегатом класса `Context`.

Больше конкретных стратегий и контекстов

Целью нашего минималистского примера было показать структуру нашего шаблона проектирования. Однако для шаблона Стратегия, возможно, требуется более серьезный пример, наглядно демонстрирующий его возможности в более реалистичном контексте.

В нашем следующем примере мы представим использование шаблона Стратегия в агентстве по подбору клоунов. В этом агентстве есть несколько различных клоунов. Клиенты звонят в него для заказа выступлений на различных мероприятиях: цирковых представлениях, днях рождения, политических собраниях. Они хотят либо конкретных клоунов, либо конкретные их выступления (сценки), либо трюки, которые они могут показать. Чтобы создать гибкую программу для такого агентства, каждая сценка или трюк будет упакована в отдельный алгоритм. Алгоритмы будут инкапсулированы, и клоуны будут делегировать свои артистические способности этим алгоритмам – стратегиям.

Клоуны

Первым делом мы создадим контекстный класс и конкретные контексты, в которых мы будем использовать различные трюки и сценки. В примере 11.6 показан наш главный класс контекста, в котором устанавливаются ссылки на интерфейсы методов стратегий. Все операции по выполнению трюков и сценок делегируются классам стратегий. В примерах 11.7 и 11.8 создаются два конкретных клоуна. Каждому из них присваиваются свои трюк и сценка, которые делегируются конкретным объектам стратегии. Для начала откройте три файла ActionScript и введите в них код из примеров с 11.6 по 11.8, сохранив каждый из них под именем из заголовка соответствующего примера. (Полное приложение состоит из примеров с 11.6 по 11.16.)

Пример 11.6. Clown.as

```
package
{

    class Clown
    {
        protected var tricks:Tricks;
        protected var skits:Skits;

        public function doTrick():void
        {
            tricks.trick();
        }

        public function doSkit():void
        {
            skits.skit();
```

```
        }
    }
}
```

Пример 11.7. Koka.as

```
package
{
    class Koka extends Clown
    {
        trace("* =>Koka<= *");
        public function Koka()
        {
            tricks = new Disappear();
            skits = new Chase();
        }
    }
}
```

Пример 11.8. Bojo.as

```
package
{
    class Bojo extends Clown
    {
        trace("* =>Koka<= *");
        public function Bojo()
        {
            tricks = new BalloonAnimals();
            skits = new FallDown();
        }
    }
}
```

Обратите внимание, что каждый из наших клоунов может выполнять только один трюк и одну сценку. Позднее мы рассмотрим возможность изменения наших стратегий, а также можно ли динамически добавлять им новые возможности. Но пока только один трюк и одна сценка для каждого клоуна.

Интерфейс и имплементации трюков

Все алгоритмы по исполнению трюков инкапсулированы у нас в конкретных подклассах стратегии. Интерфейс трюков `Tricks` (пример 11.9) описывает абстрактный метод, а подклассы наполняют его конкретными действиями. Таким образом, имплементируются все три алгоритма трюков (примеры 11.10, 11.11 и 11.12).

Пример 11.9. Tricks.as

```
package
{
    interface Tricks
```

```
    {
        function trick():void;
    }
}
```

Пример 11.10. BalloonAnimals.as

```
package
{
    // Делаем фигурки из воздушных шаров
    public class BalloonAnimals implements Tricks
    {
        public function trick():void
        {
            trace("See! It's a doggy! No, it's not an elephant!!\n")
        }
    }
}
```

Пример 11.11. Disappear.as

```
package
{
    // Заставляем что-то исчезнуть
    public class Disappear implements Tricks
    {
        public function trick():void
        {
            trace("Now you see it! Presto! It's gone!\n")
        }
    }
}
```

Пример 11.12. Juggle.as

```
package
{
    // Жонглируем шариками
    public class Juggle implements Tricks
    {

        public function trick():void
        {
            trace("Look at me juggle! Whoops!\n")
        }
    }
}
```

Применив такое общее понятие, как «трюки», мы несколько ограничили тип представлений, которые могут выполнять клоуны. В нашем случае реализации шаблона проектирования Стратегия каждый клоун может исполнять только один трюк. Мы могли бы имплементировать

класс, содержащий в себе все трюки, но это было бы слишком уж много и требовало бы от клоунов умения выполнять их все, даже если они могут выполнять только два из них.

Интерфейс и имплементации сценок

Интерфейс сценок `Skits` (пример 11.13) и все его имплементирующие классы (примеры 11.14 и 11.15) созданы нами аналогично классу трюков и их имплементаций. И ограничения здесь действуют те же.

Пример 11.13. Skits.as

```
package
{
    // Интерфейс сценок
    interface Skits
    {
        function skit():void;
    }
}
```

Пример 11.14. FallDown.as

```
package
{
    // Падает по лестнице
    public class FallDown implements Skits
    {
        public function skit():void
        {
            trace("I'm climbing up this ladder!")
            trace("Where's my banana peel?")
            trace("Whoaaaa! I'm falling!!")
            trace("Thanks for finding my banana peel!\n")
        }
    }
}
```

Пример 11.15. Chase.as

```
package
{
    // Клоуны гоняются друг за другом
    public class Chase implements Skits
    {
        public function skit():void
        {
            trace("Here I come! I'm going to get you!")
            trace("Nah! Nah! Can't catch me!\n")
        }
    }
}
```

Хотя мы и упоминали некоторые ограничения в нашем проекте, в нем все равно видно, как стратегии реализуются в имплементациях классов Tricks и Skits. Последним шагом мы попробуем запустить нашу программу.

К нам пришли клоуны!

Для данного примера в классе ClownCollege (пример 11.16) создаются два различных объекта клоунов. Они объявляются через тип Clown в соответствии с рекомендацией использовать при программировании интерфейсы, а не их имплементации, однако создаются с помощью подклассов класса Clown. У каждого из них есть свои трюки и сценки, и они готовы к представлению. Сохраните пример 11.16 под именем из его заголовка.

Пример 11.16. ClownCollege.as

```
package
{
    import flash.display.Sprite;

    public class ClownCollege extends Sprite
    {
        public function ClownCollege()
        {
            var joker:Clown=new Koka();
            joker.doTrick();
            joker.doSkit();

            var gagGrrrl:Clown=new Bojo();
            gagGrrrl.doTrick();
            gagGrrrl.doSkit();
        }
    }
}
```

После того как вы сохраните примеры с 11.6 по 11.16, откройте новый документ Flash и сохраните его как *ClownCollege.fla*. В его поле класс документа введите ClownCollege и протестируйте данное приложение. Вы должны увидеть следующий результат в окне Output:

```
* =>Koka<= *
Now you see it! Presto! It's gone!

Here I come! I'm going to get you!
Nah! Nah! Can't catch me!

* =>Bojo<= *
See! It's a doggy! No, it's not that!!
```

```
I'm climbing up this ladder!
Where's my banana peel?
Whoaaaa! I'm falling!!
Thanks for finding my banana peel!
```

Здесь показана работа алгоритмов из классов стратегии. Все их вызовы делегированы из контекстного класса `Clown` и имплементированы в конкретных классах клоунов.

Дополнительная функциональность для клоунов

Как нами отмечалось в самом начале создания приложения о клоунах с помощью шаблона проектирования Стратегия, данная его структура не очень гибкая и не позволяет динамически изменять конкретный контекст для принятия другой стратегии. Так как же нам быть, если мы хотим добавить еще одного клоуна, который может выполнять больше одного трюка, но не все из них?

К счастью, у нас есть простое решение для данной проблемы. Если мы добавим метод-установщик в контекстный класс `Clown`, то тогда станет возможным динамически добавлять стратегии в конкретные классы контекстов, т. е. клоунам. В примере 11.17 приводится обновленная версия контекстного класса `Clown` с добавленными установщиками (показаны жирным шрифтом).

Пример 11.17. Clown.as

```
package
{
    class Clown
    {
        protected var tricks:Tricks;
        protected var skits:Skits;

        public function doTrick():void
        {
            tricks.trick();
        }

        public function doSkit():void
        {
            skits.skit();
        }

        public function setTrick(addTrick:Tricks):void
        {
            tricks=addTrick;
        }

        public function setSkit(addSkit:Skits):void
```

```
        {
            tricks=addSkit;
        }
    }
}
```

Добавление нового клоуна

Наше приложение уже структурно подготовлено к простому добавлению нового конкретного класса контекста. Поэтому нам нужно лишь добавить такой подкласс и включить в него делегирование на конкретные стратегии. Пример 11.18 делает именно это. Сохраните его в файле под именем из заголовка примера.

Пример 11.18. Bubbles.as

```
package
{
    class Bubbles extends Clown
    {
        trace("* =>Bubbles<= *");
        public function Bubbles()
        {
            tricks=new Juggle();
            skits = new FallDown();
        }
    }
}
```

Добавление нового трюка

Добавить в шаблон Стратегия новую конкретную стратегию так же просто, как добавить новый конкретный класс контекста. Сохраните пример 11.19 под именем из его заголовка. Этот новый сценарий полностью повторяет все прочие конкретные стратегии, за исключением имени класса и содержания метода трюка.

Пример 11.19. BubblePants.as

```
package
{
    class BubblePants implements Tricks
    {
        public function trick():void
        {
            trace("Woo woo woo! Bubbles are coming out of my pants!\n");
        }
    }
}
```

Ревизия колледжа клоунов

Чтобы продемонстрировать, как легко добавлять новые элементы в приложение и динамически подключать новые стратегии в конкретный класс контекста, класс ClownCollege был нами отредактирован и дополнен (что показано жирным шрифтом). Во-первых, был создан новый объект конкретного класса клоунов Bubbles («пузыри»), и использованы два метода делегирования, показывающие трюки и сценку. Во-вторых, объект клоуна добавил к себе в репертуар новый трюк с помощью нового метода setTrick(). Наконец, метод doTrick() был вызван во второй раз, чтобы продемонстрировать этот новый трюк. Сохраните пример 11.20 под именем из его заголовка.

Пример 11.20. ClownCollege.as

```
package
{
    import flash.display.Sprite;

    public class ClownCollege extends Sprite
    {
        public function ClownCollege()
        {
            var joker:Clown=new Koka();
            joker.doTrick();
            joker.doSkit();

            var gagGrrrl:Clown=new Bojo();
            gagGrrrl.doTrick();
            gagGrrrl.doSkit();

            var gurgle:Clown=new Bubbles();
            gurgle.doSkit();
            gurgle.doTrick();
            gurgle.setTrick(new BubblePants());
            gurgle.doTrick();
        }
    }
}
```

Для первых двух клоунов ничего не изменилось. Однако наш новый клоун должен делать что-то новенькое, но вот что? Если посмотреть на его исходный код, то его начальными трюком и сценкой были жонглирование и падение с лестницы. Это не изменяется, поэтому его начальное выступление не должно отличаться от того, что мы уже видели.

Однако после вызова метода doTrick() у объекта клоуна gurgle («булькать») ему дается новый трюк. После повторного вызова этого метода клоун gurgle показывает уже другой трюк, что выделено с помощью жирного шрифта в следующей распечатке результатов.

```
* =>Koka<= *
Now you see it! Presto! It's gone!

Here I come! I'm going to get you!
Nah! Nah! Can't catch me!

* =>Bojo<= *
See! It's a doggy! No, it's not that!!

I'm climbing up this ladder!
Where's my banana peel?
Whoaaaa! I'm falling!!
Thanks for finding my banana peel!

* =>Bubbles<= *
I'm climbing up this ladder!
Where's my banana peel?
Whoaaaa! I'm falling!!
Thanks for finding my banana peel!

Look at me juggle! Whoops!

Woo woo woo! Bubbles are coming out of my pants!
```

Таким образом, в результате добавления установщиков в класс контекста у приложения появилась возможность динамически изменять стратегию, которую может делегировать конкретный класс контекста.

Реорганизация трюков и сценок: планирование клоунов

До того как мы перейдем к следующему примеру, нам нужно обсудить, насколько хорошо организованы подклассы стратегий для нашей текущей задачи. Помните, что, как бы замечательно или круто ни выглядело ваше приложение, хорошее планирование сделает его еще совершеннее.

Говорят, что чем больше в вашем приложении различных компонентов, тем больше в нем *гранулярности*. Она дает больше гибкости, и это хорошая сторона дела. Однако высокая гранулярность приложения приводит к большому числу классов, что добавляет головной боли по присмотру за ними.

Чтобы оптимизировать ваше приложение, особенно использующее шаблон проектирования Стратегия, в котором у вас может быть много контекстных классов и стратегий, вам нужен хороший план. Внимательно посмотрев на наш пример с клоунами, мы можем заметить, где наш исходный план был не совсем удачный.

Начинающаяся со стратегий трюков, наша структура позволяла только один трюк на клоуна. Поэтому, возможно, нам стоит реорганизовать стратегии приблизительно так, как показано в табл. 11.1.

Таблица 11.1. Стратегии

Жонглирование	Трюки с шариками	Магические трюки
Шарами	Фигуры зверушек	С картами
Кольцами	Фигуры людей	Пропадание предметов
Горящими факелами	Изображения политиков	Пропадание детей
Любыми предметами	Любые фигуры	Любая магия

Теперь вместо одного интерфейса для трюков у нас их целых три. Кроме того, появился и общий алгоритм, описывающий любой трюк в каждой категории (жонглирование любыми предметами, скручивание любых фигурок из шариков и любую магию). А для покрытия всех непредвиденных обстоятельств в классе контекста сохраняется возможность динамически добавлять новые стратегии конкретным объектам контекста.

Аналогичным образом могут быть реорганизованы и стратегии всех сценок клоунов, равно как и увеличено общее число стратегий по мере роста данного приложения.

Работа со строковыми стратегиями

Один из лучших аспектов шаблона проектирования Стратегия – его способность к изменениям и адаптации к конкретной ситуации. Данный шаблон является лучшим кандидатом для рекламного постера о композиции, где со всеми изменениями справляются легко и уверенно. Кроме того, он служит отличным средством для работы с большими и маленькими алгоритмами.

Практически в любом приложении вы можете столкнуться с повторяющимися действиями. На случай таких приложений мы решили включить в эту главу примеры нескольких маленьких алгоритмов, которые мы применяем в разных приложениях и которые приходится перестраивать каждый раз, когда мы создаем приложение, использующее их. Однако мы не хотим заключать их в большие классы, главный объект которых унаследует всю их функциональность. Лучше мы будем использовать их в композиции, что позволит нам иметь больше гибкости и применять нужную нам от них функциональность, только когда нам это потребуется.

Для нашего примера мы взяли две простые темы строковых алгоритмов и задачи сортировки, которые встречаются во многих приложениях. Сначала мы создадим несколько простых стратегий по проверке строчек. Наша первая стратегия будет смотреть, есть ли в адресе электронной почты символ «@», и не находится ли он в первой или последней позиции этой строки. Вторая стратегия будет алгоритмом проверки

паролей, переводящим всю запись в регистр строчных букв и выясняющим, совпадает ли она с правильным паролем.

Наш второй интерфейс стратегий будет демонстрировать различные возможности сортировки с помощью класса `Array` и его метода `sort()`. Изменяя параметры сортировки, вы сможете увидеть, какие константы нужно использовать при различных типах сортировки, и что при этом получается. Рисунок 11.4 показывает все связи между нашими классами.

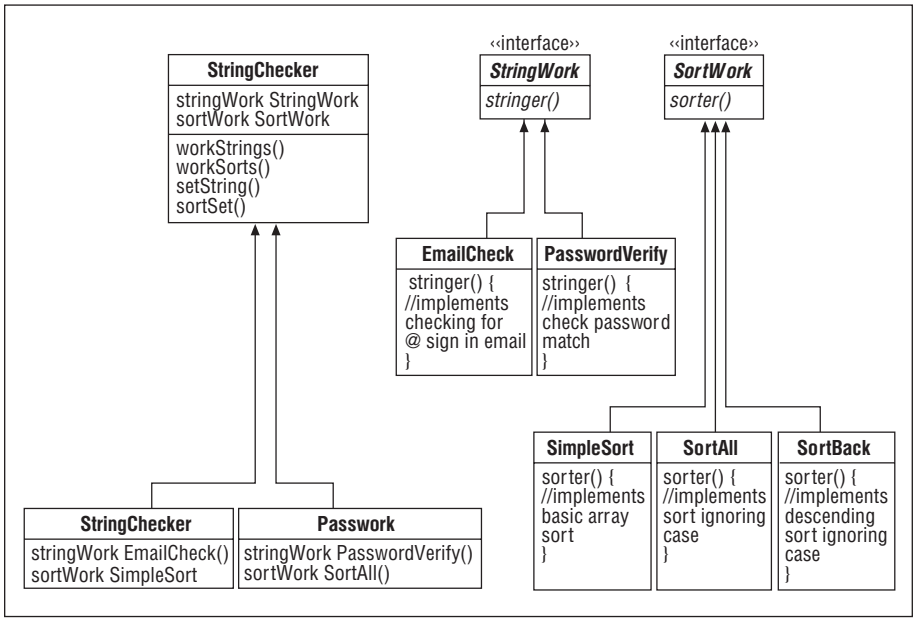


***Рис. 11.4.** Прикладное использование шаблона Стратегия*

Код нашего приложения располагается в файлах ActionScript, приведенных в примерах с 11.21 по 11.33. Нашему приложению потребуется также один файл документа Flash с классом документа для него под именем `TestStringStrategy`.

Контекст для строковых стратегий

Классы контекста в шаблоне проектирования Стратегия можно рассматривать как *клиентов* для различных стратегий, поскольку они используют их. Но иногда вы можете услышать, что клиент *присваивает* себе их поведение. То есть всякий раз, когда класс клиента делегирует запрос к классу делегата, такому как классы стратегий на рис. 11.4, он *присваивает* себе поведение или действия, генерируемые этими классами стратегий. В данном приложении мы будем рассматривать классы контекста только как клиентов классов стратегий.

Как и прочие контекстные классы, класс StringChecker из примера 11.21 создает ссылку на два интерфейса стратегий, StringWork и SortWork. Кроме того, в нем имеются методы workStrings() и workSorts(), которые будут делегировать работу стратегиям.

Пример 11.21. StringChecker.as

```
package
{
    // Контекстный класс
    class StringChecker
    {
        protected var stringWork:StringWork;
        protected var sortWork:SortWork;
        public function StringChecker():void
        {
        }

        public function workStrings(s:String):String
        {
            return stringWork.stringer(s);
        }
        public function workSorts(a:Array):Array
        {
            return sortWork.sorter(a);
        }
        public function setString(sw:StringWork):void
        {
            stringWork=sw;
        }
        public function setSort(sow:SortWork):void
        {
            sortWork=sow;
        }
    }
}
```

В дополнение к ссылкам на свойства и методы данный контекстный класс включает в себя методы-установщики setString() и setSort() для динамической установки стратегий в контекстных классах.

Далее, в примерах 11.22 и 11.23, два конкретных класса контекста расширяют исходный класс. Обратите внимание, что они наследуют ссылки на стратегии stringWork и sortWork. Эти ссылки потом используются в них для спецификации конкретных стратегий.

Пример 11.22. Checker.as

```
package
{
    // Конкретный контекст
    class Checker extends StringChecker
```

```
    {
        public function Checker()
        {
            stringWork = new EmailCheck();
            sortWork = new SimpleSort();
        }
    }
}
```

Пример 11.23. Passwork.as

```
package
{
    // Конкретный контекст
    class Passwork extends StringChecker
    {
        public function Passwork()
        {
            stringWork = new PasswordVerify();
            sortWork = new SortAll();
        }
    }
}
```

Не забывайте, что оба эти класса наследуют методы главного контекстного класса `StringChecker`; поэтому, когда вы будете тестировать работу их представителей, вы сможете использовать методы `workStrings()` и `workSorts()`.

Строковые стратегии

Классы, ассоциированные со стратегиями, гораздо проще классов их клиентов в шаблоне проектирования Стратегия. Эти интерфейсы перечисляют методы, которые будут имплементироваться в конкретных классах стратегий и наполняться конкретными действиями в форме алгоритмов. Однако в данном приложении (примеры 11.24 и 11.25) оба метода из интерфейсов требуют некоторых параметров и возвращают результаты своей работы.

Пример 11.24. StringWork.as

```
package
{
    // Стратегия
    interface StringWork
    {
        function stringer(s:String):String;
    }
}
```

Пример 11.25. SortWork.as

```
package
{
    // Стратегия
    interface SortWork
    {
        function sorter(a:Array):Array;
    }
}
```

Но даже при наличии входных параметров и возвращаемых результатов указанного типа эти интерфейсы очень просты. Гораздо более интересны их имплементации.

Стратегии проверки

Наши первые две конкретные стратегии (примеры 11.26 и 11.27) имплементируют свои операции с помощью методов класса `String`. Каждая из них представляет собой некоторый алгоритм, необходимый для работы со строками. В них нет ничего особенно мудреного, но обе они будут полезны в определенных приложениях, где приходится проверять строки.

Пример 11.26. EmailCheck.as

```
package
{
    // Конкретная стратегия
    class EmailCheck implements StringWork
    {
        public function stringer(s:String):String
        {
            var atPlace:int=s.indexOf("@");
            if (atPlace != -1 && atPlace !=0 && atPlace != (s.length-1))
            {
                return "Email address verified";
            } else
            {
                return "Email does not verify.\nMissing or
                    misplaced @ sign";
            }
        }
    }
}
```

Пример 11.27. PasswordVerify.as

```
package
{
    // Конкретная стратегия
    class PasswordVerify implements StringWork
    {
```

```
        public function stringer(s:String):String
        {
            var pwv:String=s;
            pwv=pwv.toLocaleLowerCase();
            if (pwv == "sandlight")
            {
                return "Welcome to Sandlight";
            } else
            {
                return "Your password is incorrect.
                    Please enter again.";
            }
        }
    }
}
```

Здесь вместо простого сравнения строк можно было бы использовать регулярные выражения. При применении типа данных `RegExp` и его алгоритмов возможны гораздо более сложные поиск в строках и их сравнение.

Стратегии сортировки

Алгоритмы, используемые для стратегий сортировки (примеры 11.28, 11.29 и 11.30), намного проще, чем наши простые строковые алгоритмы, примененные в двух имплементациях `StringWork`. Все эти три алгоритма используют метод `sort()` класса `Array`. Тем не менее изменяя его параметры, можно получить три различных вида сортировки.

Пример 11.28. SimpleSort.as

```
package
{
    // Конкретная стратегия
    class SimpleSort implements SortWork
    {
        public function sorter(a:Array):Array
        {
            a.sort();
            return a;


        }
    }
}
```

Пример 11.29. SortAll.as

```
package
{
    // Конкретная стратегия
    class SortAll implements SortWork
```

```
    {
        public function sorter(a:Array):Array
        {
            a.sort(Array.CASEINSENSITIVE);
            return a;


        }
    }
}
```

Пример 11.30. SortBack.as

```
package
{
    // Конкретная стратегия
    class SortBack implements SortWork
    {
        public function sorter(a:Array):Array
        {
            a.sort(Array.CASEINSENSITIVE | Array.DESCENDING);
            return a;

        }
    }
}
```

Как видите, с помощью только одного метода можно сделать довольно много. Чтобы добавить еще несколько вариантов сортировки, создайте несколько имплементаций с использованием метода `Array.sortOn()` и многомерных массивов.

Вспомогательные классы

Два вспомогательных класса помогут нам избавить наш основной тестирующий класс приложения от рассмотрения лишних деталей и позволят продемонстрировать использование идеи о структурной организации классов. Класс `ShowText` (пример 11.31) создает динамическое текстовое поле, а класс `TextShow` (пример 11.32) обеспечивает ему формат.

Пример 11.31. ShowText.as

```
package
{
    import flash.text.TextField;
    import flash.text.TextFieldType;

    class ShowText extends TextField
    {
        public function ShowText():void
        {
```

```
            this.type=TextFieldType.DYNAMIC;
            this.multiline=true;
            this.wordWrap=true;
            this.border=true;
            this.borderColor=0xcccccc;
        }
    }
}
```

Пример 11.32. TextShow.as

```
package
{
    import flash.text.TextFormat;

    class TextShow extends TextFormat
    {
        function TextShow():void
        {
            this.font="Verdana";
            this.color=0xcc0000;
            this.size=11;
            this.leading=3;
            this.leftMargin=11;
            this.rightMargin=6;
        }
    }
}
```

Не стесняйтесь изменить любой из данных вспомогательных классов согласно вашему вкусу.

Тестирование стратегий работы со строками

Наш тест реализации шаблона проектирования Стратегия (пример 11.33) использует два объекта. Первый объект создается с помощью конкретного класса контекста `Checker`, а второй – с помощью `Passwork`.

Пример 11.33. TestStringStrategy.as

```
package
{
    import flash.display.Sprite;

    public class TestStringStrategy extends Sprite
    {
        private var showText:ShowText;
        private var showText2:ShowText;
        private var textShow:TextShow;

        public function TestStringStrategy():void
        {
```

```
            doText();

            // stringer – использует конкретный контекст Checker
            var stringer:StringChecker= new Checker();
            showText.text="Stringer\n\n"+stringer.workStrings
                ("Bill@office.net");
            var friends:Array=new Array("John","Carl","aYo","Delia");
            showText.appendText("\n"+stringer.workSorts(friends));
            stringer.setString(new PasswordVerify());
            showText.appendText("\n"+stringer.workStrings("Sandlight"));

            // passer – использует конкретный контекст Passwork
            var passer:StringChecker= new Passwork();
            showText2.text="Passer\n\n"+passer.workStrings("Rumple");
            showText2.appendText("\n"+passer.workSorts(friends));
            passer.setSort(new SortBack());
            showText2.appendText("\n"+passer.workSorts(friends));
            passer.setString(new EmailCheck);
            showText2.appendText
                ("\n"+passer.workStrings("@passGuy.com"));
        }
        private function doText():void
        {
            showText=new ShowText();
            showText2=new ShowText();
            textShow=new TextShow();
            showText.defaultTextFormat = textShow;
            showText2.defaultTextFormat = textShow;
            addChild(showText),addChild(showText2);
            showText.x=50, showText2.x=270;
            showText.y=100, showText2.y=100;
            showText.width=200,showText2.width=200;
            showText.height=150,showText2.height=150;
        }
    }
}
```

Не забывайте, что вызовы методов конкретных классов `workString` и `workSorts` на самом деле делегируются классам стратегий. После создания нового документа Flash и ввода `TextStringStrategy` в его поле класса документа, протестируйте это приложение. Рисунок 11.5 демонстрирует то, что вы должны увидеть в результате.

Обратите внимание, что после динамического изменения стратегий с помощью `setString()` и `setSort()` результат работы изменился. Хотя включение таких установщиков и является опциональным для шаблона проектирования Стратегия, новый уровень гибкости, который они при этом добавляют в приложение, нельзя недооценивать.

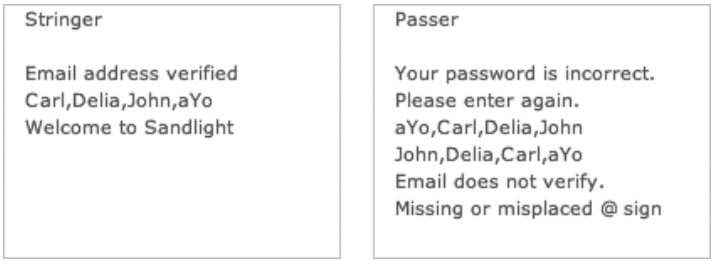


Рис. 11.5. *Результат работы TextStringStrategy*

Заключение

После рассмотрения нескольких приложений, использующих шаблон проектирования Стратегия, вы можете понять возможную область его применения. Он, несомненно, является примером использования хороших практик ООП. Наверно, самая очевидная из них – это использование композиции вместо наследования для создания и использования алгоритмов. Однако в нем используется и простое *наследование* ООП для создания конкретных классов, входящих в его состав.

Кроме того, вы можете наблюдать в нем множество способов использования *полиморфизма* при имплементации стратегий. Каждый из их интерфейсов демонстрирует применение *абстракции*, равно как и главный класс контекста. Все конкретные стратегии *инкапсулируют* в себе алгоритмы, и каждая служит делегатом для другого класса. Таким образом, мы здесь можем встретить сразу четыре базовых элемента хорошего ООП:

- Наследование
- Полиморфизм
- Абстракцию
- Инкапсуляцию

Как мы видели в данной главе в разделе, посвященном основным концепциям ООП, шаблон проектирования Стратегия демонстрирует нам применение и некоторых других практик хорошего ООП в дополнение к базовым.

Будучи хорошим примером использования фундаментальных практик ООП, шаблон Стратегия является еще и полноценным шаблоном проектирования. Иронично, что этот факт нагляднее всего проявляется при сравнении Стратегии с шаблоном проектирования Состояние, в дублировании которого его чаще всего обвиняют. Шаблон Состояние инкапсулирует состояния и контекст состояний, а шаблон Стратегия инкап-

сулирует алгоритмы. У них разные задачи, но эти задачи маскируются их практически идентичными диаграммами классов. Только при их глубоком понимании и некоторой практике использования разница между ними обнаруживается в полной мере. Тогда же становится ясна и область применения каждого из шаблонов Состояние и Стратегия, призванных решать разные задачи. Поскольку по определению шаблона проектирования Стратегия его областью использования являются повторяющиеся задачи, мы теперь ясно видим, что он полностью удовлетворяет этому своему основному критерию.

V

Множественные шаблоны

Множественная личность – в известном смысле совершенно нормальное явление.

Джордж Герберт Мид (Описание того, что является нормальным для программистов, пытающихся использовать множественные шаблоны проектирования.)

Язык может быть абстрагирован, язык может быть использован как прекрасный код в поэзии, описывающий нюансы и множества оттенков смысла, он подобен музыке. В нем столько возможностей! Он также является редукцией прозы и, следовательно, может быть математическим или очень, очень абстрактным.

Джим Джармуш

Каждое дисциплинированное усилие вознаграждается многократно.

Джим Рон

Жаль, что невозможно двигаться в двух направлениях одновременно. Но нам не позволено иметь множественные личности.

Гор Видал (Описание работы взаимоисключающих радиокнопок.)

Две главы этой последней части книги представят вам две модели, которые реализуют больше, чем просто одиночные шаблоны проектирования. Во-первых, мы рассмотрим Модель-Представление-Контроллер

(МПК). МПК использует шаблон Наблюдатель (глава 8) для хранения данных в своей Модели, в Представлении применяется шаблон Композиция (глава 6) для отображения состояния всей модели, а Контроллер применяет шаблон Стратегия (глава 11) для взаимодействия с пользователем. Наша имплементация МПК на ActionScript 3.0 будет следовать списку шаблонов проектирования, предложенному «бандой четырех». В МПК возможно использование и других шаблонов проектирования, таких как Фабрика и Декоратор.

Вторым нашим множественным шаблоном проектирования будет шаблон Симметричный заместитель, использующий в своей работе Шаблонный метод (глава 9) и шаблон Заместитель («Proxy»), который не был описан в данной книге. Для доставки видео в метод заместителя, находящийся в классе плеера, мы использовали Flash Media Server 2, вместо того чтобы создавать отдельные классы для заместителя и реального плеера. Однако шаблон Симметричный заместитель, как и любой хороший шаблон, достаточно устойчив, чтобы выдержать такие изменения без утраты своей собственной основной структуры или функций.

Глава 12 «Шаблон Модель-Представление-Контроллер»

Глава 13 «Шаблон Симметричный заместитель»

12

Шаблон Модель–Представление–Контроллер

Согласно стандартной модели миллиарды лет назад какая-то маленькая квантовая флуктуация, наверно, чуть меньшая плотности материи, возможно, именно там, где мы сейчас сидим, стала причиной начала формирования нашей галактики.

Сет Ллойд

Всякий раз мы смотрим на вещи не только с другой стороны, но и другими глазами – поэтому и считаем, что они переменились.

Блез Паскаль

Основным признаком контролера является отрицание, то есть я не могу видеть его признаки в себе.

Кит Миллер

Что такое шаблон Модель–Представление–Контроллер (МПК)?

Модель-Представление-Контроллер (МПК) – это составной шаблон, или несколько шаблонов, работающих совместно, для создания сложных приложений. Шаблон МПК особенно часто используется для создания интерфейсов приложений и, как ясно из его имени, состоит из трех частей.

Модель. Содержит данные приложения и его логику работы, зависящую от состояния приложения.

Представление. Представляет состояние приложения на экране и пользовательский интерфейс.

Контроллер. Обрабатывает ввод данных от пользователя, чтобы изменить состояние приложения.

Мощь шаблона МПК может быть откровенно приписана разделению функций между тремя его элементами без перекрытия зон их ответственности. Давайте рассмотрим эти зоны.

Модель

Модель ответственна за управление состоянием приложения. Логика работы модели выполняет две очень важные задачи: она отвечает на запросы информации о состоянии приложения и выполняет запросы по его изменению.

Представление

Представление, можно сказать, является лицом приложения. Именно через него пользователь общается с приложением. У одного приложения может быть множество представлений, являющихся как входом, так и выходом для информации пользователя. Например, в случае портативного проигрывателя музыки, такого как iPod, его экран является его представлением. Его кнопки, контролирующие воспроизведение мелодий, также являются его представлениями. На экране показываются имя текущей песни, ее продолжительность, название альбома и так далее – все, что относится к текущему состоянию этого устройства. Представлению совсем не обязательно быть визуальным. В случае цифрового музыкального плеера музыка, звучащая в наушниках, также является представлением. И, например, нажатие на одну из кнопок может выдавать некоторое звуковое подтверждение этого действия в наушники в форме звука нажатия. Изменение уровня звука также влияет на результирующий звук в наушниках. Таким образом, аудиовыход соответствует состоянию данного приложения.

Контроллер

Хотя термин контроллер и подразумевает некоторый интерфейс для контроля за приложением, в шаблоне МПК контроллер не содержит никаких элементов интерфейса пользователя. Как упоминалось ранее, элементы пользовательского интерфейса, обеспечивающие ввод от него информации, находятся в элементе *представление*. Контроллер же определяет, как представление реагирует на действия пользователя.

Пусть, например, наш цифровой музыкальный плеер имеет в своем *представлении* кнопки «громкость вверх» и «громкость вниз», и уровень громкости для данного устройства будет численной переменной,

изменяющейся от 0 до 10. Модель, которая содержит в себе эту переменную, нуждается в некоторой логике работы приложения по ее изменению. И именно контроллер определяет, насколько поднимется или опустится уровень громкости звука при одиночном нажатии на вышеуказанные кнопки. Контроллер может сказать модели поднять уровень на 0.5, или на 1.0, или на любое другое запрограммированное значение. Таким образом, контроллер является некоторой специфической имплементацией, определяющей, как приложение реагирует на действия пользователя.

Хотя каждый из элементов триады МПК имеет собственную, уникальную зону ответственности, они не работают в полной изоляции друг от друга. Фактически чтобы быть единым шаблоном каждому из этих элементов нужно взаимодействовать с одним или более других элементов. Мы рассмотрим такое взаимодействие в следующем разделе.

Взаимодействие между элементами МПК

Каждый элемент в шаблоне МПК взаимодействует с другими его элементами строго определенными способами. Эти взаимодействия порождают последовательности событий, которые обычно запускаются некоторыми действиями пользователя приложения. Описание такой последовательности приведено ниже:

1. Пользователь приложения взаимодействует с элементом интерфейса пользователя (например, нажимает на кнопку в представлении).
2. Представление посылает соответствующее событие контроллеру, чтобы он решил, как его нужно обработать.
3. Контроллер производит изменения в модели на основе своего решения о том, как следует обработать это событие.
4. Модель информирует представление об изменении своего состояния.
5. Представление считывает информацию о новом состоянии и обновляет себя.

Рисунок 12.1 показывает графическое представление всех описанных взаимодействий между элементами МПК. Стрелки указывают направления взаимодействий.

Данная модель взаимодействия элементов МПК очень простая. В некоторых случаях контроллер может напрямую говорить представлению сделать некоторые изменения. Это происходит тогда, когда действия пользователя не требуют изменений в модели, но изменяют визуальное представление. Например, представьте себе процесс, когда пользователь выбирает мелодию для проигрывания на нашем цифровом музыкальном плеере. Он выбирает ее из списка на экране, используя для этого кнопки прокрутки списка. Представление будет информировать контроллер о нажатии кнопок прокрутки списка вверх или вниз, но контроллер не сообщит об этом модели. Он укажет непосредственно

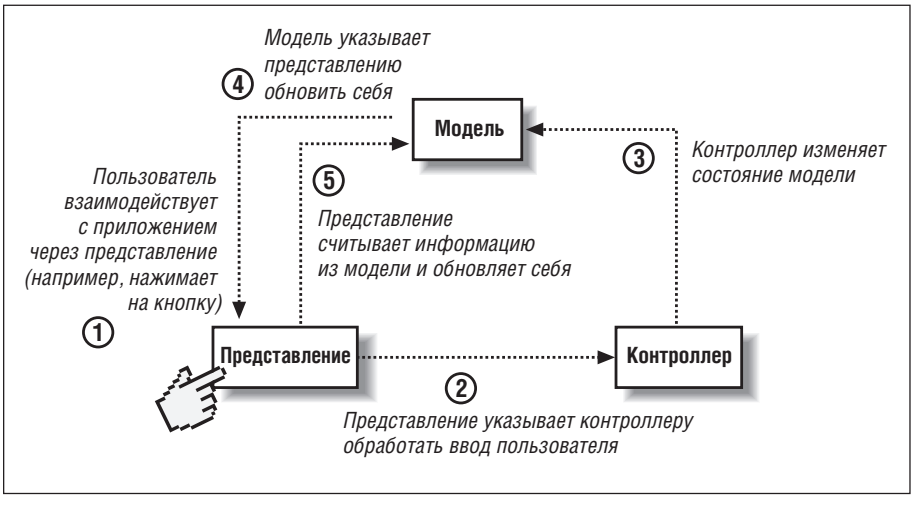


Рис. 12.1. *Направления взаимодействий между элементами шаблона МПК*

представлению прокрутить список мелодий в нужном направлении. Действия пользователя не требуют в данном случае изменений в модели. Однако, когда пользователь в конце концов выберет мелодию для воспроизведения, контроллер изменит модель, чтобы отразить этот выбор в текущей песне.

Изменения в модели не всегда инициируются взаимодействием с пользователем. Модель может обновляться и на основе других событий. Например, представьте себе приложение для отображения текущей стоимости акций. Его модель будет содержать в себе все эти цены. Однако цены на акции постоянно меняются, и модель может установить себе таймер для периодического обновления цен акций с соответствующего веб-сервера. Таким образом, когда бы модель ни обновила данные о цене акций, она уведомит представление об изменении своего состояния.

Еще одна возможность шаблона проектирования МПК состоит в том, что каждая модель может иметь более одного своего представления. Например, настройка уровня громкости в нашем цифровом музыкальном плеере может быть показана на экране соответствующим индикатором, а также в уровне громкости звука в наушниках плеера. Экран и звук в наушниках являются представлениями для состояния данного устройства.

Взгляните еще раз на рис. 12.1 и обратите внимание на направления стрелок взаимодействий. Они показывают, кто инициирует данное взаимодействие между элементами. Чтобы один из элементов МПК мог взаимодействовать с другим его элементом, он должен о нем знать, и в нем должна быть ссылка на него.

Давайте рассмотрим модель, представление и контроллер как три различных класса. Давайте обсудим, каким из этих классов нужны ссылки на другие классы.

Модель

Необходима ссылка на *представления*

Представление

Необходимы ссылки на *модель* и *контроллер*

Контроллер

Необходима ссылка на *модель*

Мы начали с утверждения, что МПК является составным шаблоном, состоящим из нескольких шаблонов проектирования. И вы можете спросить, а где же эти внутренние шаблоны? К ним мы сейчас и перейдем. Итак, основным преимуществом использования шаблона МПК являются слабые связи между его тремя элементами. Они позволяют нам, например, привязать несколько представлений к одной модели и заменить модели и контроллеры без нарушения работы остальных элементов. Однако элементы шаблона МПК должны содержать в себе ссылки на другие его элементы для взаимодействия между ними. Как же мы можем называть это слабыми связями? Вот где нам на помощь приходят шаблоны Наблюдатель, Стратегия и Композиция.

Внутренние шаблоны МПК

Мы уже говорили, что в МПК одна модель может быть ассоциирована с несколькими представлениями. Следовательно, она должна всех их информировать об изменениях, происходящих в ней. Причем модель должна делать это, не зная всех деталей о своих представлениях и о том, сколько именно представлений должно быть изменено. Данная распространенная проблема взаимодействия лучше всего решается с помощью *шаблона Наблюдатель* (глава 8).

Представления, связанные с моделью, могут быть весьма сложными, состоящими из многих окон или панелей, содержащих различные элементы интерфейса пользователя. Например, такие элементы этого интерфейса, как кнопки, текстовые поля, списки, слайдеры и т. д., могут быть сгруппированы вместе на одной из переключаемых панелей, которая в свою очередь может быть частью окна или другой переключаемой панели. Даже каждая кнопка или группа кнопок может быть представлением. Равно как и набор текстовых полей. Так что было бы очень удобно рассматривать панель или окно, содержащие несколько простых представлений, тем же способом, что и любое другое представление. Именно в этом нам поможет *шаблон Композиция* (глава 6). Почему он может быть полезен нам в данном контексте? Если все представления будут вложены друг в друга, как должно быть в шаблоне

Композиция, процесс их обновления будет для нас значительно проще, поскольку он будет автоматически спускаться вниз к дочерним представлениям. Создание сложных представлений значительно упростится, если нас не будут волновать вопросы о том, как обеспечить все вложенные представления информацией об обновлениях.

Представления являются средством внешнего отображения состояния модели. Они делегируют все события от интерфейса пользователя к контроллеру. Следовательно, в контроллере должен содержаться алгоритм, как обрабатывать действия пользователя в каждом конкретном представлении. Такое делегирование позволяет инкапсулировать имплементацию действий конкретных элементов пользовательского интерфейса в терминах изменения модели МПК. Мы можем легко заменять контроллеры для одного и того же представления, чтобы получать требуемое нам поведение. Это идеальный контекст для имплементации *шаблона Стратегия*.

Давайте рассмотрим, как каждый из указанных шаблонов имплементируется в МПК, на его минималистском примере.

Минималистский пример шаблона МПК

Этот простой пример будет следить за тем, какая клавиша была нажата последней на клавиатуре. При очередном таком нажатии он будет менять модель и информировать представление о необходимости обновления. Представление будет использовать обычную панель Output Flash для печати *кода символа* нажатой клавиши. Код символа – это числовое значение из текущего набора символов. Данный пример призван прояснить для нас, как шаблоны Наблюдатель, Стратегия и Композиция могут быть интегрированы внутри МПК.

Модель как конкретная рассылка в шаблоне Наблюдатель

Рассмотрим взаимоотношения между моделью и представлением как взаимоотношения между рассылкой и наблюдателем (см. главу 8). В модели должен быть имплементирован интерфейс рассылки, являющийся частью шаблона Наблюдатель. К счастью, в ActionScript 3.0 есть встроенные классы, делающие это, и мы можем уведомлять наблюдателя об изменениях, используя модель уведомлений о событиях.

Класс рассылки событий в ActionScript 3.0

Класс `EventDispatcher` имплементирует интерфейс `IEventDispatcher`. Среди прочих данный интерфейс определяет следующие методы, нужные нам для рассылки из шаблона Наблюдатель (см. детальное описание их параметров в документации по ActionScript 3.0).

```
addEventListener(type:String,
    listener:Function,
    useCapture:Boolean = false,
    priority:int = 0,
    useWeakReference:Boolean = false):void

removeEventListener(type:String,
    listener:Function,
    useCapture:Boolean = false):void

dispatchEvent(event:Event):Boolean
```

Чтобы модель могла работать как конкретная рассылка из шаблона Наблюдатель, она должна имплементировать интерфейс `IEventDispatcher`. Однако простейшим способом создать свой класс с возможностью рассылки уведомлений является расширение встроенного класса `EventDispatcher`.

Наблюдатель регистрирует свои методы обработки получаемых уведомлений о событиях от объекта типа `EventDispatcher` с помощью метода `addEventListener()`.

Модель

Наша модель будет хранить код символа последней нажатой клавиши в своем свойстве. В ней должны быть имплементированы методы *установщик* и *получатель* для этого свойства, чтобы представление и контроллер могли получать его значение и модифицировать его. Давайте сначала определим интерфейс для нашей модели (пример 12.1).

Пример 12.1. IModel.as

```
package
{
    import flash.events.*;

    public interface IModel extends IEventDispatcher
    {
        function setKey(key:uint):void
        function getKey():uint
    }
}
```

Интерфейс `IModel` из примера 12.1 расширяет интерфейс `IEventDispatcher` и определяет дополнительно два метода для получения и установки кода символа последней нажатой клавиши. Поскольку интерфейс `IModel` расширяет `IEventDispatcher`, любой класс, который будет имплементировать данный интерфейс, должен будет имплементировать все методы, определенные в обоих этих интерфейсах. Класс `Model`, показанный в примере 12.2, имплементирует интерфейс `IModel`.

Пример 12.2. Model.as

```
package
{
    import flash.events.*;

    public class Model extends EventDispatcher implements IModel
    {
        private var lastKeyPressed:uint = 0;

        public function setKey(key:uint):void
        {
            this.lastKeyPressed = key;
            dispatchEvent(new Event(Event.CHANGE)); // разослать событие
        }

        public function getKey():uint
        {
            return lastKeyPressed;
        }
    }
}
```

Класс `Model` **расширяет класс** `EventDispatcher`**, в котором уже имплементирован интерфейс** `IEventDispatcher`**. Обратите внимание, что функция** `dispatchEvent()` **вызывается внутри метода** `setKey()`**. Она посылает событие** `CHANGE` **всем зарегистрированным наблюдателям, когда значение переменной** `lastKeyPressed` **изменилось внутри метода** `setKey()`**.**

Контроллер как конкретная стратегия

Взаимоотношения между контроллером и представлением повторяют отношения между стратегией и классом контекста в шаблоне Стратегия. Каждый контроллер будет конкретной стратегией, имплементирующей свое поведение, описанное в интерфейсе стратегии.

Контроллер

Для нашего минималистского примера от контроллера требуется обработка события о нажатии клавиши на клавиатуре. `IKeyboardInputHandler` **будет нашим интерфейсом стратегии (пример 12.3) и будет определять единственный метод** `keyPressHandler()`**.**

Пример 12.3. IKeyboardInputHandler.as

```
package
{
    import flash.events.*;

    public interface IKeyboardInputHandler
    {
        function keyPressHandler(event:KeyboardEvent):void
```

```
    }
}
```

Классом конкретного контроллера будет класс `Controller` (пример 12.4), который имплементирует интерфейс `IKeyboardInputHandler`.

Пример 12.4. Controller.as

```
package
{
    import flash.events.*;

    public class Controller implements IKeyboardInputHandler
    {
        private var model:IModel;

        public function Controller(aModel:IModel)
        {
            this.model = aModel;
        }

        public function keyPressHandler(event:KeyboardEvent):void
        {
            model.setKey(event.charCode); // изменить модель
        }
    }
}
```

Обратите внимание, что конструктор контроллера получает объект модели как параметр. Это необходимо, поскольку именно контроллер инициирует взаимодействие с моделью, как показано на рисунке 12.1. Следовательно, он должен хранить в себе ссылку на модель.

Метод `keyPressHandler()` получает событие от интерфейса пользователя (в данном случае `KeyboardEvent`) как параметр и решает, как его обработать. В данном примере он просто устанавливает значение в модели кода последней нажатой клавиши в код символа этой клавиши.

Представление как конкретный наблюдатель и контекст из шаблона Стратегия

Представление – это, вероятно, самый сложный элемент в шаблоне МПК. Оно должно одновременно играть роли элементов шаблонов Наблюдатель и Стратегия, что формирует базис взаимоотношений между моделью и контроллером. Класс `View`, показанный в примере 12.5, имплементирует представление для нашего минималистского примера.

Пример 12.5. View.as

```
1 package
2 {
```

```
 3     import flash.events.*;
 4     import flash.display.*;
 5
 6     public class View
 7     {
 8         private var model:IModel;
 9         private var controller:IKeyboardInputHandler;
10
11         public function View(aModel:IModel, oController:
                        IKeyboardInputHandler,target:Stage)
12         {
13             this.model = aModel;
14             this.controller = oController;
15
16             // регистрируем для получения уведомлений от модели
17             model.addEventListener(Event.CHANGE, this.update);
18
19             // регистрируем для получения информации о нажатых клавишах
20             target.addEventListener(KeyboardEvent.KEY_DOWN,  onKeyPress);
21
22         }
23
24         private function update(event:Event):void
25         {
26             // получить данные из модели и обновить представление
27             trace(model.getKey());
28         }
29
30         private function onKeyPress(event:KeyboardEvent):void
31         {
32             // делегировать контроллеру (стратегии) управление
33             controller.keyPressHandler(event);
34         }
35     }
36 }
```

Поскольку представление инициирует взаимодействия с моделью и контроллером, как показано на рис. 12.1, оно должно содержать ссылки на них. Объекты модели и контроллера передаются конструктору представления. Кроме того, представлению из нашего примера нужна ссылка на объект сцены, чтобы зарегистрировать себя на получение событий о нажатии клавиш.

Помимо создания пользовательского интерфейса класс `View` выполняет еще несколько важных задач. Он регистрируется на получение событий от модели об обновлении ее состояния и делегирует контроллеру обработку действий пользователя. В нашем примере представление не имеет визуальных объектов на сцене приложения, но отображает состояние модели на панели Output. Ему требуется получение событий о нажатии клавиш, для чего его метод `onKeyPress()` регистрируется на получение событий `KEY_DOWN` от сцены приложения (строка 20). Кроме того, в нем

регистрируется метод update() на получение события CHANGE от модели (строка 16). При получении такого события об изменении данный метод считывает из модели код символа последней нажатой клавиши и печатает его в выходном окне приложения, используя функцию trace().

Построение триады МПК

Мы уже рассмотрели индивидуальные имплементации всех трех элементов, входящих в шаблон МПК. Однако нам еще нужен его клиент, инициализирующий каждый из этих элементов и создающий итоговую конструкцию. Это не потребует больших действий; нужно лишь создать объекты модели, представления и контроллера. Пример 12.6 демонстрирует класс документа Flash, создающий все наши элементы МПК.

Пример 12.6. Main.as (класс документа для минималистского примера)

```
package
{
    import flash.display.*;
    import flash.events.*;

    /**
    *    класс Main
    *    @ назначение: Класс документа
    */
    public class Main extends Sprite
    {
        public function Main()
        {
            var model:IModel = new Model();
            var controller:IKeyboardInputHandler = new Controller(model);
            var view:View = new View(model, controller, this.stage);
        }
    }
}
```

После своего создания модель, контроллер и представление начнут взаимодействие друг с другом, и приложение заработает. Нажатие на клавиши клавиатуры будет приводить к печати кода символа на панели Output.

Вам нужно будет отключить клавиши быстрого вызова при тестировании данного приложения (иначе интерфейс пользователя Flash продолжит перехватывать их и обрабатывать). Чтобы это сделать, выберите Disable Keyboard Shortcuts (Отключить клавиши быстрого вызова) из меню Control (Контроль) при работе данного приложения.

Обратите внимание, что объект модели передается контроллеру, а объекты модели и контроллера передаются представлению. Мы можем легко заменить модель и контроллер другими при условии, что они им-

плементируют интерфейсы `IModel` и `IKeyboardInputHandler` соответственно. В наше приложение могут быть добавлены дополнительные представления, поскольку между моделью и представлениями существует связь, подобная связи между рассылкой и наблюдателями. Модель даже не знает о своих представлениях, так как они сами должны регистрироваться на получение сообщений об изменениях от модели. В этом и состоит красота шаблона МПК; модель, представление и контроллер являются отдельными, слабосвязанными элементами, позволяющими очень гибко ими пользоваться.

Вложенные представления как листья и узлы шаблона Компоновщик

Как мы уже говорили, представление – это, наверно самая сложная по структуре часть шаблона МПК, поскольку она участвует в имплементациях и шаблона Стратегия, и шаблона Наблюдатель. Но наше представление станет сейчас еще сложнее, поскольку оно будет имплементировать в себе еще и третий шаблон, Компоновщик (см. главу 6). Такая имплементация представления имеет смысл, только когда оно состоит из множества вложенных друг в друга пользовательских интерфейсов, содержащих несколько представлений. Вложенные композитные представления обладают тем преимуществом, что обновление может автоматически распространяться вниз по их древовидной структуре. Кроме того, композитные представления могут динамически создавать и удалять дочерние представления в зависимости от состояния приложения и режима работы с ним пользователя. Хорошим примером сложного пользовательского интерфейса является панель *инспектора свойств* в среде разработки Flash. Данный инспектор свойств является контекстно-зависимым и добавляет или удаляет элементы пользовательского интерфейса в зависимости от выбранного элемента на сцене.

Оконечные и составные представления

Первым шагом мы создадим оконечные и составные классы для представления. Они должны рассматриваться как абстрактные и не должны создавать своих представителей, а только подклассы. Первый класс оконечного компонента представления показан в примере 12.7.

Пример 12.7. ComponentView.as

```
package
{
    import flash.errors.IllegalOperationError;
    import flash.events.Event;
    import flash.display.Sprite;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class ComponentView extends Sprite
    {
```

```
        protected var model:Object;
        protected var controller:Object;

        public function ComponentView(aModel:Object, aController:Object = null)

        {
            this.model = aModel;
            this.controller = aController;
        }

        public function add(c:ComponentView):void
        {
            throw new IllegalOperationError(«add operation not supported»);
        }

        public function remove(c:ComponentView):void
        {
            throw new IllegalOperationError(«remove operation not supported»);

        }

        public function getChild(n:int):ComponentView
        {
            throw new IllegalOperationError(«getChild operation not supported»);

            return null;
        }

        // Абстрактный метод (должен быть замещен в подклассе)
        public function update(event:Event = null):void {}
    }
}
```

Класс `ComponentView`, показанный в примере 12.7, определяет собой абстрактный интерфейс компонента представления. Он очень похож на классический класс оконечного компонента из главы 6 с некоторыми важными изменениями. Данный класс хранит в себе ссылки на модель и на представление, которые передаются его конструктору. Не все представления обрабатывают действия пользователя, и оконечное представление может быть создано при помощи ссылки только на модель. Поэтому параметр контроллера для данного конструктора, `aController`, имеет значение по умолчанию, равное `null`. Кроме того, обратите внимание, что класс `ComponentView` является расширением класса `Sprite`. Это разумно, поскольку большинство представлений должно рисовать элементы пользовательского интерфейса на сцене приложения. Мы можем использовать свойства и методы этого встроенного класса `Sprite` для рисования различных объектов и добавления их в список отображения.

Описанный здесь метод `update()` должен рассматриваться как абстрактный метод. Оконечные, листовые представления, подклассы данного класса `ComponentView`, должны замещать и имплементировать его, что-

бы обновлять состояние своего пользовательского интерфейса. Данный метод является обработчиком сообщений, перехватывающим все уведомления об обновлении от модели. По этой причине у него есть параметр Event, но со значением по умолчанию, равным null, что позволяет вызывать данный метод и без передачи ему параметра события. Это используется при первоначальном создании интерфейса пользователя в начальном состоянии, что продемонстрирует наш следующий пример.

Класс CompositeView расширяет ComponentView и замещает его методы работы с дочерними представлениями. В примере 12.8 мы имплементируем из них только метод add() для простоты.

Пример 12.8. CompositeView.as

```
package {

    import flash.events.Event;

    // Абстрактный класс (должен быть наследован, а не реализован)
    public class CompositeView extends ComponentView
    {
        private var aChildren:Array;

        public function CompositeView(aModel:Object,aController:Object = null)
        {
            super(aModel, aController);
            this.aChildren = new Array();
        }

        override public function add(c:ComponentView):void
        {
            aChildren.push(c);
        }

        override public function update(event:Event = null):void
        {
            for each (var c:ComponentView in aChildren)
            {
                c.update(event);
            }
        }
    }
}
```

Обратите внимание на замещенный метод update() в классе CompositeView из примера 12.8. Он вызывает такой же метод во всех своих детях. Следовательно, вызов его у корневого узла композитного представления спустится вниз и обойдет всю структуру дерева компонентов, обновив все представления. Давайте создадим подклассы CompositeView и ComponentView, создадим вложенные представления и посмотрим, как это все работает.

Создание вложенных представлений

Чтобы проиллюстрировать работу с вложенными представлениями, мы создадим составное представление и два его дочерних оконечных компонента, как показано на рис. 12.2.

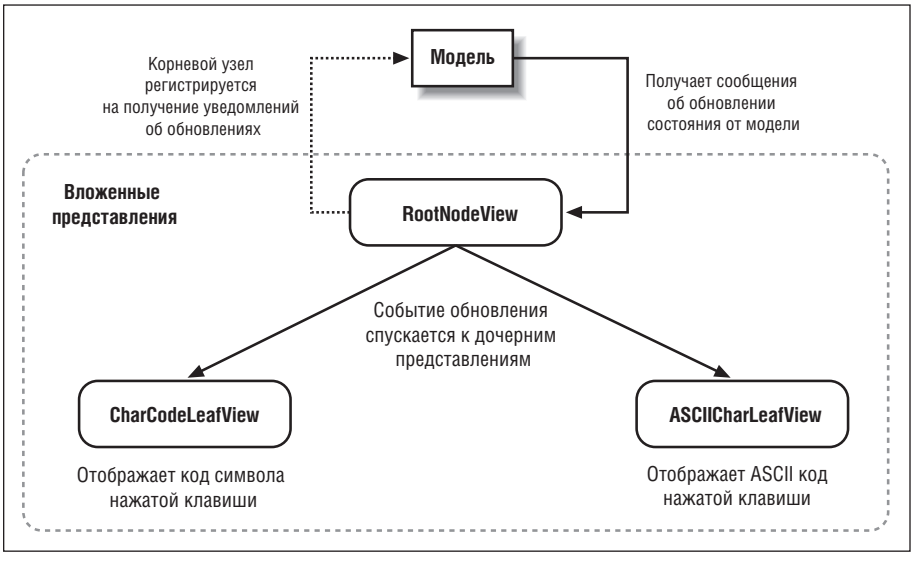


***Рис. 12.2.** Структура вложенных представлений минималистского примера*

Сначала мы создадим составное корневое представление `RootNodeView`. Оно будет получать события о вводе с клавиатуры от сцены. Оно также будет регистрироваться на получение уведомлений об обновлениях от модели. У составного корневого представления будут два дочерних оконечных представления, `CharCodeLeafView` и `AsciiCharLeafView`. Первое из них будет печатать *код символа* последней нажатой клавиши, а второе – *ASCII символ*, соответствующий такому коду. Давайте первым реализуем составное представление, показанное в примере 12.9.

Пример 12.9. RootNodeView.as

```
package
{
    import flash.events.*;
    import flash.display.*;

    public class RootNodeView extends CompositeView
    {
        public function RootNodeView (aModel:IModel,
                                       aController:Object=null,target:Stage)

        {
            super(aModel, aController);
```

```
            // регистрируем для получения уведомлений о нажатии клавиш
            target.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPress);

        }

        private function onKeyPress(event:KeyboardEvent):void
        {
            // делегируем контроллеру (стратегии) управление
            (controller as IKeyboardInputHandler).keyPressHandler(event);
        }
    }
}
```

Класс `RootNodeView`, показанный в примере 12.9, является подклассом класса `CompositeView` (пример 12.8). Он не рисует никакого пользовательского интерфейса, он просто следит за всеми событиями нажатия клавиш клавиатуры и делегирует контроллеру их обработку. Обратите внимание на использование выражения `super` в его конструкторе. Оно необходимо для вызова конструктора его суперкласса. Теперь мы можем создать два оконечных представления (примеры 12.10 и 12.11).

Пример 12.10. CharCodeLeafView.as

```
package
{
    import flash.events.Event;

    public class CharCodeLeafView extends ComponentView
    {
        public function CharCodeLeafView(aModel:IModel, aController:Object = null)
        {
            super(aModel, aController);
        }

        override public function update(event:Event = null):void
        {
            // получить данные из модели и обновить представление
            trace(model.getKey());
        }
    }
}
```

Пример 12.11. AsciiCharLeafView.as

```
package
{
    import flash.events.Event;

    public class AsciiCharLeafView extends ComponentView
    {
        public function AsciiCharLeafView(aModel:IModel,aController:Object = null)
        {
```

```
            super(aModel, aController);
        }

        override public function update(event:Event = null):void
        {
            // получить данные из модели и обновить представление
            trace(String.fromCharCode(model.getKey()));
        }
    }
}
```

Оба класса листовых представлений `CharCodeLeafView` (пример 12.10) и `AsciiCharLeafView` (пример 12.11) являются подклассами `ComponentView` (пример 12.8). Обратите внимание, что они не получают какой-либо информации от пользовательского интерфейса. Следовательно, оба этих представления могут быть созданы без передачи их конструкторам ссылки на контроллер. В данном случае она останется по умолчанию равной `null`.

Теперь нам осталось только построить с помощью клиента триаду элементов МПК с вложенными представлениями. Клиент зарегистрирует корневой узел составного представления на получение уведомлений об обновлениях состояния от модели.

Построение структуры вложенных представлений

Построение вложенных представлений полностью идентично созданию структуры из шаблона Компоновщик. Нам нужно визуализировать наше представление как дерево (т. е. перевернутое дерево) и использовать метод `add()` в его составном узле, чтобы добавить дочерние представления.

```
var model:IModel = new Model();
var controller:IKeyboardInputHandler = new Controller(model);

// составное представление
var rootView:CompositeView = new RootNodeView(model,controller, this.stage);

// добавляем вложенные классы представления
rootView.add(new CharCodeLeafView(model));
rootView.add(new AsciiCharLeafView(model));

// регистрируем представителя для получения уведомлений от модели
model.addEventListener(Event.CHANGE, rootView.update);
```

Заметьте, что только корневой узел представления регистрируется на получение событий об обновлении состояния модели. Поскольку это составной узел, событие спускается ко всем его дочерним узлам. Теперь при любом нажатии клавиши один из дочерних узлов напечатает код ее символа, а второй – ASCII символ этой клавиши. Хотя наша структура вложенных представлений для данного минималистского примера до-

вольно проста, она может хорошо работать с вложенными представлениями, имеющими много компонентов.

В нашем минималистском примере клиент сам создавал структуру вложенных представлений. Однако ее метод построения может быть включен в класс корневого компонента `RootNodeView`, что еще больше инкапсулирует ее имплементацию. Тогда корневое представление сможет динамически добавлять и удалять дочерние представления в зависимости от состояния приложения или режима работы с ним пользователя. Это даст пользовательскому интерфейсу приложения несколько очень мощных возможностей.

Ключевые особенности шаблона МПК

Основной полезной чертой шаблона Модель-Представление-Контроллер является гибкость, с которой он позволяет строить приложения с пользовательским интерфейсом. Шаблон МПК позволяет разделить модель, представление и контроллер приложения и использует преимущества шаблонов Наблюдатель, Стратегия и Композиция для структурирования своих элементов.

- Шаблон МПК состоит из трех элементов, называемых модель, представление и контроллер, между которыми разделяются зоны ответственности в приложении с графическим пользовательским интерфейсом.
- Взаимодействие между моделями и представлениями строится на основе отношений между конкретной рассылкой и конкретным наблюдателем в шаблоне Наблюдатель.
- Взаимодействие между представлениями и контроллерами строится на основе отношений между контекстом и конкретной стратегией в шаблоне Стратегия.
- У одной модели может быть множество представлений.
- Представления могут быть вложенными друг в друга с использованием шаблона Композиция для создания сложных интерфейсов пользователя с простым процессом их обновления.

Ключевые концепции ООП в шаблоне МПК

Ключевой концепцией в шаблоне МПК является *слабосвязанность*, которая ослабляет взаимозависимости между различными элементами этого шаблона. Например, взаимодействие по типу рассылка-наблюдатели между моделью и ее представлениями позволяет первой работать совершенно независимо от того, как ее состояние отображается в пользовательском интерфейсе. Только на представление возлагается ответственность подписаться на получение обновлений состояния

модели и соответственно обновлять элементы интерфейса пользователя. Модель при этом функционирует совершенно независимо, в полном счастливом неведении о числе и разновидностях своих представлений. В приложение без нарушения этой связи могут быть введены и вложенные представления. Сообщения об обновлениях будут просачиваться вниз ко всем вложенным представлениям, если корневое представление будет подписано на их рассылку от модели. Благодаря всему этому шаблон МПК является легко расширяемым.

Кроме того, каждый элемент в МПК жестко следует принципу четкого распределения ответственностей и имеет свою, четко определенную роль. Модель занимается состоянием, представление отображает состояние, а контроллер обрабатывает действия пользователя. Это позволяет легко заменять любые элементы без изменений во всех прочих. Например, если нам потребовалось другое поведение для какого-то элемента в представлении, мы просто меняем для него соответствующий контроллер. Все это делает шаблон МПК очень легко настраиваемым и позволяет многократно использовать имеющиеся наработки.

Пример: карты погоды

Национальное управление по атмосферным и океаническим явлениям (National Oceanic and Atmospheric Administration (NOAA)), являющееся подразделением министерства торговли США, ведет во Всемирной паутине «Геостационарный спутниковый сервер» («Geostationary Satellite Server») – *http://www.goes.noaa.gov*. На данном сайте публикуются спутниковые снимки США, включая Пуэрто-Рико, Аляску и Гавайи. Мы будем использовать эти снимки (все они совершенно бесплатные) для разработки приложения простой карты погоды с помощью шаблона МПК. Мы будем применять встроенные компоненты пользовательского интерфейса из Flash CS3 для разработки элементов наших представлений.

Для первой версии нашего приложения, показанного на рис. 12.3, мы позволим пользователю выбирать регион для отображения (Восточное побережье, Западное побережье, Пуэрто-Рико, Аляска и Гавайи) при помощи выпадающего списка. После этого приложение будет загружать последний видимый спутниковый снимок соответствующего региона.

Этот пример продемонстрирует нам, как использовать встроенные компоненты Flash CS3 для имплементации пользовательского интерфейса в каждом из представлений. Он также покажет полезность использования вложенных представлений и их автоматических обновлений. Давайте создадим модель для нашего учебного приложения.

Модель

Модель как элемент шаблона МПК содержит в себе данные приложения и его состояние, включая логику по управлению ими. Состоянием

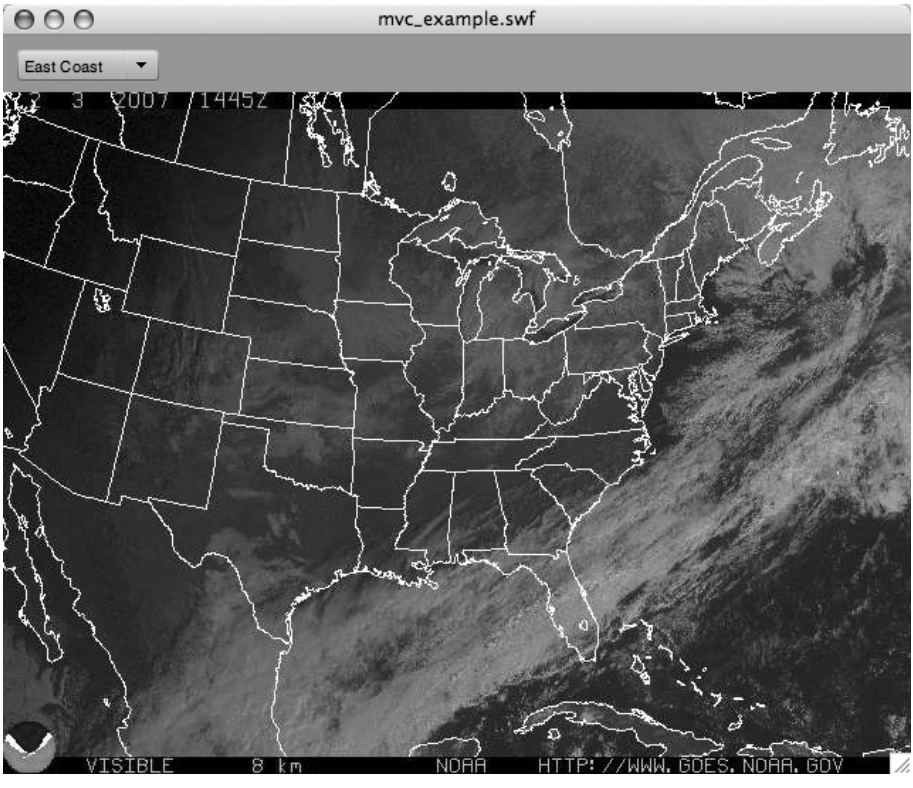


Рис. 12.3. *Учебное приложение «Карта погоды», показывающее Восточное побережье США*

нашего приложения будет тот регион, который пользователь выбрал для отображения. Это может быть один из пяти регионов, доступных с указанного ранее веб-сайта. Данные приложения – это адреса URL спутниковых изображений для всех указанных регионов.

Логика работы приложения должна позволять пользователю управлять его состоянием. То есть у пользователя должна быть возможность выбирать текущий регион. Элементу представления, который будет рисовать выпадающий список, потребуется как-то получать список всех доступных регионов и текущий выбранный регион (чтобы отметить его в этом списке). Кроме того, представлению, которое будет показывать спутниковые снимки, понадобится доступ к адресу URL для текущего выбранного региона. Давайте же разработаем интерфейс нашей модели (пример 12.12) с учетом всех этих требований.

Пример 12.12. IModel.as (Интерфейс модели для примера с картами погоды)

```
package
{
    import flash.events.*;
```

```
    public interface IModel extends IEventDispatcher
    {
        function getRegionList():Array
        function getRegion():uint
        function setRegion(index:uint):void
        function getMapURL():String
    }
}
```

Следующим шагом нам нужно имплементировать данный интерфейс и реализовать нашу модель (пример 12.13).

Пример 12.13. Model.as (Модель для примера с картами погоды)

```
package {

    import flash.events.*;

    public class Model extends EventDispatcher implements IModel
    {

        protected var aRegions:Array;
        protected var chosenRegion:uint;

        protected var aImageURLs:Array;

        public function Model()
        {
            this.aRegions = new Array(
                    “East Coast”,
                    “West Coast”,
                    “Puerto Rico”,
                    “Alaska”,
                    “Hawaii”);
            this.aImageURLs = new Array(
                    “http://www.goes.noaa.gov/GIFS/ECVS.JPG”,
                    “http://www.goes.noaa.gov/GIFS/WCVS.JPG”,
                    “http://www.goes.noaa.gov/GIFS/PRVS.JPG”,
                    “http://www.goes.noaa.gov/GIFS/ALVS.JPG”,
                    “http://www.goes.noaa.gov/GIFS/HAVS.JPG”);
            this.chosenRegion = 0;
        }

        public function getRegionList():Array
        {
            return aRegions;
        }

        public function getRegion():uint
        {
            return this.chosenRegion;
        }
```

```
        public function setRegion(index:uint):void
        {
            this.chosenRegion = index;
            this.update();
        }

        public function getMapURL():String
        {
            return this.aImageURLs[chosenRegion];
        }

        protected function update():void
        {
            dispatchEvent(new Event(Event.CHANGE)); // разослать событие
        }
    }
}
```

Обратите внимание на метод `update()` в классе `Model` из примера 12.13. Он рассылает событие `CHANGE` для всех зарегистрированных наблюдателей. Данный метод вызывается при любом изменении состояния приложения.

Как для разработчика для вас может быть соблазнительным хранить список всех регионов для простоты сразу в представлении. Но это крайне снизит гибкость нашего проекта и возможности по повторному использованию имеющихся наработок. Например, добавление нового региона потребует тогда изменений и в модели, и в этом представлении. Всегда лучше придерживаться проверенной практики четко очерченных зон ответственности для каждого из трех элементов шаблона МПК. Данные приложения должны храниться и получаться только из модели.

Контроллер

В нашем приложении его пользователь может контролировать только элемент представления с выпадающим списком. Интерфейс для соответствующего контроллера показан в примере 12.14.

Пример 12.14. ICompInputHandler.as

```
package
{
    public interface ICompInputHandler
    {
        function compChangeHandler(index:uint):void
    }
}
```

Его имплементация показана в примере 12.15.

Пример 12.15. Controller.as (Контроллер для примера с картами погоды)

```
package
{
    public class Controller implements ICompInputHandler
    {

        private var model:Object;

        public function Controller(aModel:IModel)
        {
            this.model = aModel;
        }

        public function compChangeHandler(index:uint):void {
            (model as IModel).setRegion(index); // обновить модель
        }
    }
}
```

Представления

Наше приложение, отображающее карты погоды, состоит из двух представлений. Первое из них – это элемент интерфейса пользователя, который содержит выпадающий список для выбора региона. Второе представление отображает соответствующий спутниковый снимок.

Мы будем использовать встроенные компоненты ComboBox (выпадающий список) и UILoader (загрузчик) из Flash CS3 для реализации элементов интерфейса пользователя в каждом из представлений. Компоненты – это мувиклипы с параметрами, позволяющими вам изменять их отображение и поведение. С их помощью можно создавать Flash-приложения с нужным видом и поведением без разработки заново таких элементов пользовательского интерфейса, как слайдеры и кнопки. Для нашего приложения с картами погоды нам нужно перетащить компоненты ComboBox и UILoader с панели компонентов Components на панель библиотеки разработчика Library для нашего документа Flash. Обе данные панели доступны из меню Windows во Flash CS3. Обратите внимание, что пока не нужно помещать эти компоненты на сцену. Мы поместим их туда во время исполнения программы прямо из ActionScript. Компонент ComboBox использует в себе еще несколько дополнительных ресурсов. Ваша панель Library будет выглядеть, как изображенная на рис. 12.4, после перетаскивания на нее компонентов ComboBox и UILoader.

Теперь мы можем разработать представления, содержащие выпадающий список и изображения карт погоды для нашего приложения. Представление с выпадающим списком будет составным и содержать представление с картой как одного из своих детей.

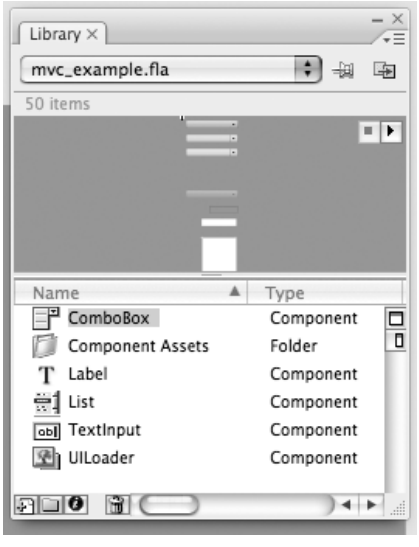


***Рис. 12.4.** Панель Library с компонентами ComboBox и UILoader*

Представление с выпадающим списком

Класс `CBView` (пример 12.16) является подклассом `CompositeView` (пример 12.8) и отображает компонент `ComboBox`.

Пример 12.16. CBView.as

```
1 package
2 {
3     import flash.events.Event;
4     import fl.controls.ComboBox;
5
6     public class CBView extends CompositeView
7     {
8
9         private var cb:ComboBox;
10
11        public function CBView(aModel:IModel,aController:
                                ICompInputHandler= null)
12    {
13            super(aModel, aController);
14
15            // получить имена регионов от модели
16            var aRegions:Array = (model as IModel).getRegionList();
17
18            // нарисовать выпадающий список регионов
19            cb = new ComboBox();
20            for (var i:uint = 0; i < aRegions.length; i++)
21            {
```

```
22                  cb.addItem( { label: aRegions[i], data:i } );
23              }
24              update();
25              addChild(cb);
26
27              // зарегистрировать для получения изменений в выпадающем списке
28              cb.addEventListener(Event.CHANGE, this.changeHandler);
29          }
30
31          override public function update(event:Event = null):void
32          {
33              // получить данные от модели и обновить представление
34              cb.selectedIndex = (model as IModel).getRegion();
35              super.update(event);
36          }
37
38          private function changeHandler(event:Event):void
39          {
40              // делегируем контроллеру (стратегии) управление
41
42              (controller as ICompInputHandler).compChangeHandler
                (ComboBox(event.target).selectedItem.data);
43          }
44      }
45  }
```

Класс `CBView` (пример 12.16) получает список имен всех регионов от модели (строка 16) и передает их всех в компонент. В строке 24 вызывается метод `update()` без параметров. Он считывает текущий выбранный регион из модели и обновляет выпадающий список. После чего он добавляет его к списку отображения сцены (строка 25) и регистрирует функцию `changeHandler()` на получение сообщений об изменениях от компонента выпадающего списка (строка 28). Заметьте, что поскольку это композитное представление, замещенный метод `update()` должен вызывать себя и в суперклассе (строка 35), чтобы обновления спустились и до дочерних представлений.

Представление с картой

Класс `MapView` (пример 12.17) является подклассом `ComponentView` (пример 12.7) и использует для отображения компонент `UILoader`.

Пример 12.17. MapView.as

```
1  package
2  {
3      import flash.events.Event;
4      import fl.containers.UILoader;
5
6      public class MapView extends ComponentView
7      {
```

```
 8          private var uiLoader:UILoader;
 9
10          public function MapView(aModel:IModel, aController:Object = null)
11          {
12              super(aModel, aController);
13
14              uiLoader = new UILoader();
15              uiLoader.scaleContent = false;
16              update();
17              addChild(uiLoader);
18          }
19
20          override public function update(event:Event = null):void
21          {
22              // получить данные от модели и обновить представление
23              uiLoader.source = (model as IModel).getMapURL();
24          }
25      }
26  }
```

Класс `MapView` (пример 12.17) загружает и показывает спутниковый снимок и больше никак не взаимодействует с пользователем. Как и в представлении с выпадающим списком, здесь метод `update()` вызывается без параметров в конструкторе класса (строка 16) для загрузки изображения региона по умолчанию. Присвоение адреса URL свойству `source` компонента `UILoader` приводит к загрузке указанного изображения. В отличие от представления с выпадающим списком, данное представление с картой является оконечным и не может иметь детей. Следовательно, его замещенный метод `update()` не должен вызывать свой метод в суперклассе.

Построение триады МПК

Теперь нам осталось только создать модель и контроллер и построить композитную структуру представлений, добавив представление с картой в качестве дочернего к представлению с выпадающим списком. Следующие выражения должны быть выполнены со стороны класса документа Flash для данного приложения.

```
var model:IModel = new Model();
var controller:ICompInputHandler= new Controller(model);

// составное представление
var view:CompositeView = new CBView(model, controller);
view.x =  view.y = 10;
addChild(view);

// добавляем подкласс в представление
var map:ComponentView = new MapView(model);
view.add(map);
```

```
map.x = 0
map.y = 40;
addChild(map);

// регистрируем представление на получение обновлений от модели
model.addEventListener(Event.CHANGE, view.update);
```

Обратите внимание, что только представление с выпадающим списком, которое является корневым узлом композитной структуры представлений, регистрируется на получение сообщений об обновлении от модели. Потом они будут спущены вниз к его дочернему представлению с картой.

Установка самообновления в модели

Изменения в состоянии модели не всегда инициируются действиями пользователя. На получаемых спутниковых снимках вы могли заметить отметки времени, показывающие, что эти изображения обновляются каждые 15 минут. Следовательно, мы можем добавить к нашей модели простой 15-минутный таймер, который будет сообщать ей, что пришло время перезагрузить спутниковый снимок. Нам нужно добавить следующие команды в конце конструктора класса `Model` из примера 12.13 (не забудьте сперва импортировать класс `flash.utils.Timer`).

```
var updateTimer:Timer = new Timer(1000 * 60 * 15);
updateTimer.addEventListener(«timer», timerHandler);
updateTimer.start();
```

Следующий обработчик событий таймера нужно добавить в класс `Model`. Этот обработчик `timeHandler()` подписывается на сообщения `TimeEvent`, которые будут посылаться каждые 15 минут. Метод `timeHandler()` будет вызывать метод `update()`, чтобы послать сообщение о необходимости обновления всем зарегистрированным наблюдателям.

```
public function timerHandler(event:TimerEvent):void {
    this.update();
}
```

Расширенный пример: инфракрасные карты погоды

Чтобы продемонстрировать, как шаблон МПК позволяет гибко расширять и повторно использовать свои модели, представления и контроллеры, мы расширим наш пример с картами погоды. Вы могли обратить внимание, что на сайте «Геостационарного спутникового сервера» (*http://www.goes.noaa.gov*) имеется три вида спутниковых снимков. Что если мы захотим дать пользователю возможность выбирать, какой из снимков просматривать: обычный или инфракрасный? Какие изменения потребуются для того, чтобы расширить наше погодное приложение новой возможностью показывать инфракрасные спутниковые снимки?

Сначала нам потребуется добавить еще одно представление с переключаемыми кнопками выбора типа спутникового снимка, обычного или инфракрасного, как показанное на рис. 12.5.

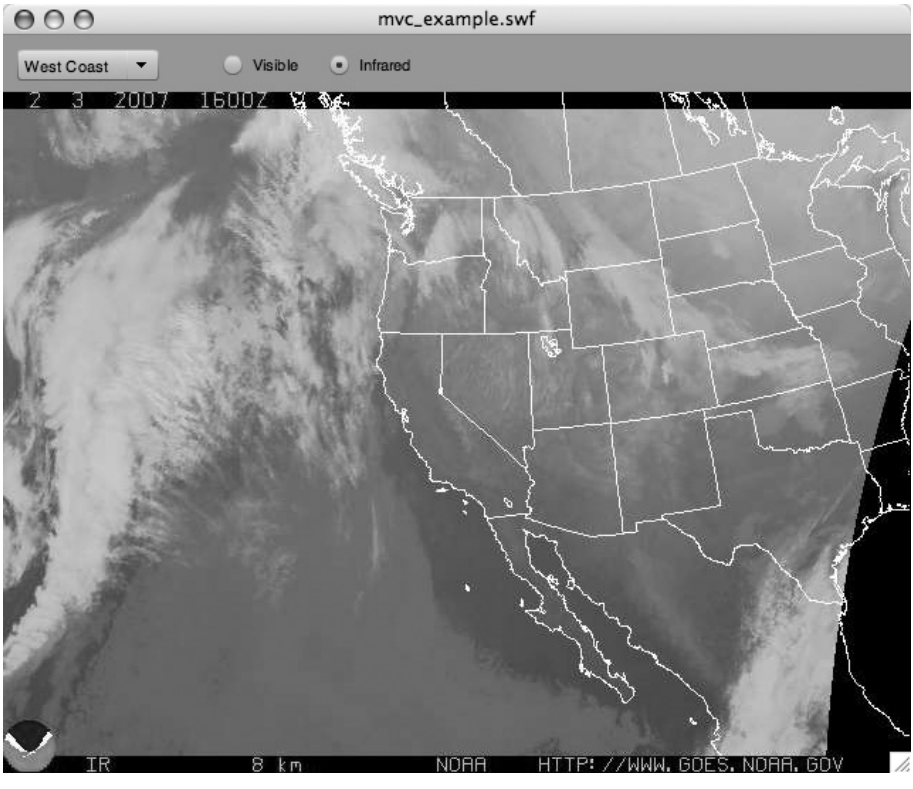


Рис. 12.5. *Расширенный пример, показывающий инфракрасный снимок Западного побережья*

Нам потребуется также создать новый контроллер для обработки действий пользователя с этим новым элементом представления. В нашей модели придется хранить больше данных, поскольку нам понадобятся пять дополнительных адресов URL для инфракрасных снимков каждого региона. Нам придется также обновить логику работы нашего приложения для обработки новых данных. Сможем ли мы добавить все эти новые возможности, не изменяя существующий код? Помогут ли нам гибкость и возможности повторного использования кода шаблона МПК без нарушения его работы?

Давайте рассмотрим, какие изменения нам потребуются в нашей модели.

Добавление новой модели

Вместо того чтобы модифицировать текущую модель для показа обычных карт погоды (пример 12.13), мы расширим ее, включив в нее новые данные и информацию о состояниях. Нам потребуется добавить еще одно свойство для хранения выбранного типа карт для показа (обычная или инфракрасная). И нам придется дополнить логику приложения для работы с новыми данными. Пример 12.18 показывает новый интерфейс модели, в котором определяются новые методы, необходимые для реализации новых свойств.

Пример 12.18. INewModel.as

```
package
{
    import flash.events.*;

    public interface INewModel extends IModel
    {
        function getMapTypeList():Array
        function getMapType():uint
        function setMapType(index:uint):void
    }
}
```

Интерфейс `INewModel` расширяет интерфейс `IModel` (пример 12.12) и определяет новые методы, необходимые для получения списка изображений в форме массива, включая методы установщики и получатели для текущего выбранного типа карты (инфракрасная или обычная). Пример 12.19 показывает класс новой модели `NewModel`, который расширяет предыдущий класс `Model`, чтобы имплементировать новый интерфейс модели `INewModel`.

Пример 12.19. NewModel.as

```
package {

    public class NewModel extends Model implements INewModel {

        protected var aMapTypes:Array;
        protected var chosenMapType:uint;

        protected var aIRImageURLs:Array;

        public function NewModel() {
            this.aIRImageURLs = new Array(
                    "http://www.goes.noaa.gov/GIFS/ECIR.JPG",
                    "http://www.goes.noaa.gov/GIFS/WCIR.JPG",
                    "http://www.goes.noaa.gov/GIFS/PRIR.JPG",
```

```
                "http://www.goes.noaa.gov/GIFS/ALIR.JPG",
                "http://www.goes.noaa.gov/GIFS/HAIR.JPG");
            this.aMapTypes = new Array(
                "Visible",
                "Infrared");
            this.chosenMapType = 0;
        }

        public function getMapTypeList():Array {
            return aMapTypes;
        }

        public function getMapType():uint {
            return this.chosenMapType;
        }

        public function setMapType(index:uint):void {
            this.chosenMapType = index;
            this.update();
        }

        override public function getMapURL():String {
            switch(chosenMapType) {
                case 1:
                    return this.aIRImageURLs[chosenRegion];
                    break;
                default:
                    return this.aImageURLs[chosenRegion];
                    break;
            }
        }
    }
}
```

Метод `getMapURL()` был замещен, поскольку возвращаемый адрес URL спутникового изображения теперь зависит от выбранного его типа. Обратите внимание, что мы не модифицировали существующий код, а только расширили существующий класс модели `Model`, чтобы имплементировать новые требования ее интерфейса.

Добавление нового контроллера

Новый контроллер (пример 12.20) нужен нам для обработки ввода от представления, содержащего выбор типа изображений. Мы можем для него имплементировать тот же интерфейс (пример 12.14).

Пример 12.20. MapTypeController.as

```
package
{
    public class MapTypeController implements ICompInputHandler {
```

```
        private var model:Object;

        public function MapTypeController(oModel:INewModel)
        {
            this.model = oModel;
        }

        public function compChangeHandler(index:uint):void
        {
            (model as INewModel).setMapType(index); // обновить модель
        }
    }
}
```

Добавление нового представления

Мы добавим также в наше приложение новое представление (пример 12.21), содержащее две переключаемые кнопки, которые позволят пользователю сделать выбор между обычными и инфракрасными изображениями. В этом представлении будет использоваться встроенный компонент для таких кнопок RadioButton из Flash CS3. Не забудьте перетащить его с панели компонентов Components на панель библиотеки Library вашего документа Flash.

Пример 12.21. RBView.as

```
package
{
    import flash.events.*;
    import fl.controls.RadioButton;
    import fl.controls.RadioButtonGroup;

    public class RBView extends ComponentView
    {
        private var rbList:Array = new Array();
        private var rbGrp:RadioButtonGroup;

        public function RBView(aModel:INewModel,aController:ICompInputHandler)
        {
            super(aModel, aController);
            // получить имена регионов от модели
            var aMapTypes:Array = model.getMapTypeList();
            // создать переключатель типов карт
            rbGrp = new RadioButtonGroup(«Map Type»);
            for (var i:uint = 0; i < aMapTypes.length; i++)
            {
                var rb:RadioButton = new RadioButton();
                rb.label = aMapTypes[i];
                rb.value = i;
                rb.group = rbGrp;
                rb.x = i * 75;
```

```
                addChild(rb);
                rbList.push(rb);
            }
            update(); // установить тип по умолчанию
            // зарегистрировать для получения изменений типа карты
            rbGrp.addEventListener(MouseEvent.CLICK, this.changeHandler);
        }

        override public function update(event:Event = null):void
        {
            // получить данные от модели и обновить представление
            var index:uint = (model as INewModel).getMapType();
            rbList[index].selected = true;
            super.update(event);
        }

        private function changeHandler(event:Event):void
        {
            // делегируем контроллеру (стратегии) управление изменением типа карты
            controller.compChangeHandler(event.target.selection.value);
        }
    }
}
```

Класс `RBView` (пример 12.21) является подклассом класса `ComponentView` (пример 12.7) и реализует пару переключаемых кнопок выбора. Компонент `RadioButton` должен использоваться в группе из по крайней мере двух своих представителей, так что только одна из кнопок группы может быть выбрана в каждый момент времени (взаимоисключающий выбор). Как и в предыдущих имплементациях представлений, здесь метод `update()` вызывается без параметров в конструкторе класса сразу после реализации всех компонентов. Это сделано для того, чтобы компонент пользовательского интерфейса отображал текущий выбор по умолчанию, полученный из модели. Теперь мы можем собрать воедино все элементы шаблона МПК и реализовать нужное композитное представление.

Построение триады МПК

Структура вложенных представлений нашего расширенного приложения для показа карт погоды изображена на рис. 12.6.

Следующие команды должны быть выполнены из класса документа Flash. Здесь только объект корневого представления `CBView` (рис. 12.6) регистрируется на получение сообщений об обновлении состояния модели. Далее эти сообщения будут просачиваться вниз ввиду особенностей реализации шаблона Композиция.

```
var model:INewModel = new NewModel(); // новая модель
var controller:ICompInputHandler = new Controller(model);
```

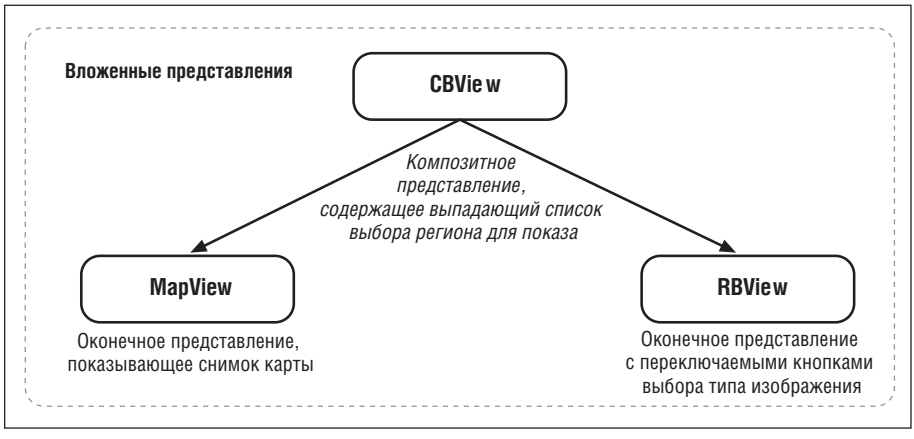


***Рис 12.6.** Структура вложенных представлений расширенного приложения для показа карт погоды*

```
// представление выбора региона
var view:CompositeView = new CBView(model, controller);
view.x =  view.y = 10;
addChild(view);

// добавляем представление карты как дочерний узел
var map:ComponentView = new MapView(model);
view.add(map);
map.x = 0
map.y = 40;
addChild(map);

// контроллер для обработки типа карты
var mapTypeController:ICompInputHandler = new MapTypeController(model);

// добавляем переключатель типа карт как дочерний узел
var mapTypeView:ComponentView = new RBView(model, mapTypeController);
view.add(mapTypeView);
mapTypeView.x =  150;
mapTypeView.y = 10;
addChild(mapTypeView);

// регистрируем представление на получение обновлений от модели
model.addEventListener(Event.CHANGE, view.update);
```

Итак, чтобы расширить наше исходное приложение показа карт погоды, мы добавили в него новое представление и новый контроллер и дополнили модель. Однако при этом мы нигде не меняли существующий код. Таким образом, все, что говорят о шаблоне МПК эксперты, было

наглядно продемонстрировано нами в данном учебном примере. Все элементы МПК слабо связаны между собой, что позволяет нам легко добавлять в него новые элементы или заменять их без нарушения работы существующего кода. Кроме того, реализация вложенных представлений с использованием шаблона Композиция позволяет нам переконфигурировать внешний вид и интерфейс нашего приложения, а также процесс получения обновлений от клиента всеми представлениями без каких-либо изменений в них.

Пример: машинки

В данном примере с помощью шаблона МПК мы разработаем простые гонки машинок, управляемые с клавиатуры. Акцент будет сделан на разработку произвольных представлений с использованием ActionScript вместо применения встроенных компонентов. Мы также увидим, как простая смена контроллера превратит машинку, контролируемую с клавиатуры, в машинку-преследователя. Для разработчиков простых игр это будет весьма полезный пример. Рисунок 12.7 показывает внешний вид финальной версии нашего учебного приложения.

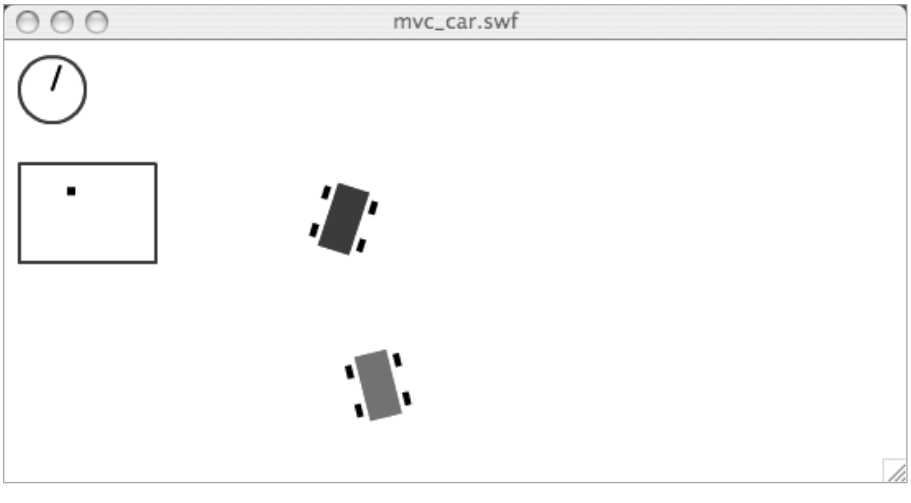


Рис. 12.7. *Пример игры с новыми представлениями и преследующей машинкой*

Модель

Первым делом мы разработаем интерфейс модели для нашей машинки. Нам нужно будет устанавливать ее положение на сцене и как-то ее поворачивать. Кроме того, мы объявим методы для установки цвета машинки. Пример 12.22 показывает нам такой интерфейс.

Пример 12.22. ICar.as

```
package
{
    import flash.geom.Point;
    import flash.events.IEventDispatcher;

    public interface ICar extends IEventDispatcher    {
        function setLoc(pt:Point):void;
        function getLoc():Point;
        function setColor(color:uint):void;
        function getColor():uint;
        function addToRotationAngle(nAngle:int):void;
        function getRotation():int;
    }
}
```

Класс `CarModel`**, показанный в примере 12.23, имплементирует интерфейс** `ICar`**. Наша машинка будут двигаться прямо вперед с постоянной скоростью. Обратите внимание, как изменяется положение машинки по таймеру.**

Пример 12.23. CarModel.as

```
package
{
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.Timer;

    public class CarModel extends EventDispatcher implements ICar
    {
        protected var nSpeed:Number; // скорость в пикселах в кадр
        protected var nRotation:Number; // поворот машинки в градусах
        protected var ptLoc:Point; // текущее местоположение
        protected var carColor:uint; // цвет

        public function CarModel()
        {
            nSpeed = 3;
            nRotation = 0;
            ptLoc = new Point(0, 0); // начинаем в 0,0
            carColor = 0x000000; // по умолчанию цвет черный

            // ставим таймер на обновление местоположения 20 раз в секунду
            var carMoveTimer:Timer = new Timer(1000 / 20);
            carMoveTimer.addEventListener(«timer», doMoveCar);
            carMoveTimer.start();
        }

        public function setLoc(pt:Point):void
        {
```

```
            ptLoc = pt;
        }

        public function getLoc():Point
        {
            return ptLoc;
        }

        public function setColor(color:uint):void
        {
            this.carColor = color;
        }

        public function getColor():uint
        {
            return carColor;
        }

        public function getRotation():int
        {
            return nRotation;
        }

        // добавляем значение к углу поворота
        public function addToRotationAngle(nAngle:int):void
        {
            nRotation += nAngle;
        }

        // перемещаем машинку
        private function doMoveCar(event:TimerEvent):void
        {
            var newLocOffset:Point = Point.polar(nSpeed, nRotation * Math.PI / 180);
            ptLoc.x += newLocOffset.x; // смещение по x
            ptLoc.y += newLocOffset.y; // смещение по y
            this.update();
        }

        protected function update():void
        {
            dispatchEvent(new Event(Event.CHANGE)); // разослать событие
        }
    }
}
```

Контроллер

Наш контроллер будет обрабатывать ввод с клавиатуры и изменять состояние модели, немного поворачивая машинку. Пример 12.24 показывает интерфейс `IKeyboardInputHandler` **для такого контроллера.**

Пример 12.24. IKeyboardInputHandler.as

```
package
{
    import flash.events.*;

    public interface IKeyboardInputHandler
    {
        function keyPressHandler(event:KeyboardEvent):void
    }
}
```

Класс `RHController`, показанный в примере 12.25, имплементирует данный интерфейс `IKeyboardInputHandler`. Он реагирует на ввод с клавиатуры и по нажатию клавиш со стрелкой вправо или влево определяет, куда именно нужно повернуть машинку. Это хороший пример контроллера, принимающего решение, как именно и насколько отреагировать на действия пользователя. В данном примере стрелки влево или вправо поворачивают нашу машинку на 8 градусов соответственно против или по часовой стрелке.

Пример 12.25. RHController.as

```
package
{
    import flash.events.*;
    import flash.ui.*;

    public class RHController implements IKeyboardInputHandler
    {
        private var model:ICar;

        public function RHController(aModel:ICar)
        {
            this.model = aModel;
        }

        public function keyPressHandler(event:KeyboardEvent):void
        {
            switch (event.keyCode)
            {
                case Keyboard.LEFT :
                    model.addToRotationAngle(-8);
                    break;
                case Keyboard.RIGHT :
                    model.addToRotationAngle(8);
                    break;
            }
        }
    }
}
```

Представления

Сейчас мы имплементируем два вложенных представления: одно для получения ввода с клавиатуры, а другое для отображения машинки на сцене. Класс представления для получения ввода с клавиатуры `KeyboardInputView`, показанный в примере 12.26, является композитным представлением. Он регистрирует свой обработчик событий нажатия клавиш для всей сцены и делегирует контроллеру их обработку.

Пример 12.26. KeyboardInputView.as

```
package
{
    import flash.events.*;
    import flash.display.*;

    public class KeyboardInputView extends CompositeView
    {
        public function KeyboardInputView(aModel:ICar,
                                          aController:IKeyboardInputHandler,
target:Stage)
        {
            super(aModel, aController);
            target.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPress);
        }

        protected function onKeyPress(event:KeyboardEvent):void
        {
            (controller as IKeyboardInputHandler).keyPressHandler(event);
        }
    }
}
```

Класс `CarView`, показанный в примере 12.27, является оконечным представлением. Он рисует машинку, используя назначенный для нее цвет. В своем методе `update()` он считывает текущее положение машинки из модели и рисует ее на соответствующем месте и с заданным поворотом. Интересным аспектом данного представления является то, что его положение изменяется. Ведь представления не обязательно должны быть подобны классическим элементам пользовательского интерфейса, таким как кнопки, статичные изображения и т. д. Они могут быть любыми отражениями текущего состояния модели.

Пример 12.27. CarView.as

```
package {

    import flash.geom.*;
    import flash.events.*;
```

```
    public class CarView extends ComponentView {

        public function CarView(aModel:ICar, aController:Object = null) {

            super(aModel, aController);

            // рисуем корпус
            graphics.beginFill(model.getColor());
            graphics.drawRect(-20, -10, 40, 20);
            graphics.endFill();
            // рисуем шины
            drawTire(-12, -15);
            drawTire(12, -15);
            drawTire(-12, 15);
            drawTire(12, 15);

            // обновляем машинку
            this.update();
        }

        private function drawTire(xLoc:int, yLoc:int) {
            graphics.beginFill(0x000000); // черный цвет
            graphics.drawRect(xLoc - 4, yLoc - 2, 8, 4);
            graphics.endFill();
        }

        override public function update(event:Event = null):void {
            // получить данные от модели и обновить представление
            var ptLoc:Point = (model as ICar).getLoc();
            this.x = ptLoc.x;
            this.y = ptLoc.y;
            this.rotation = (model as ICar).getRotation();
        }
    }
}
```

Создание машинки

Как и в предыдущих примерах, построение триады МПК будет очень простым. Мы создадим вложенные представления с объектом класса `KeyboardInputView` в качестве корневого узла и объектом класса `CarView` в качестве дочернего узла. Код клиента, показанный в примере 12.28, создает все элементы МПК и структуру вложенных представлений.

Пример 12.28. Main.as (класс документа для примера машинок)

```
1 package {
2
3     import flash.display.*;
```

```
 4     import flash.events.*;
 5     import flash.geom.*;
 6
 7     /**
 8     *    класс Main
 9     *    @ назначение: класс Документа
10     */
11     public class Main extends Sprite {
12
13         public function Main() {
14
15             var carModel:ICar = new CarModel();
16             carModel.setLoc(new Point(200,200));
17             carModel.setColor(0x0000FF); // синий
18
19             var carController:IKeyboardInputHandler =
                                              new RHController(carModel);
20
21             // представление ввода с клавиатуры (составное)
22             var kbInputView:CompositeView = new KeyboardInputView(carModel,
                                                    carController, this.stage);
23
24
25             // представление машинки (составное)
26             var car:ComponentView = new CarView(carModel);
27             kbInputView.add(car);
               // добавляем представление машинки в kbInputView
28
29             addChild(car);
30
31             // подключаемся к обновлению ввода с клавиатуры
32
33             carModel.addEventListener(Event.CHANGE, kbInputView.update);
34
35         }
36     }
37 }
```

Теперь наша машинка должна двигаться по сцене и поворачиваться при нажатии клавиш со стрелками вправо и влево. Обратите внимание, что у нас нет проверки пресечения границы сцены, и наша машинка может с нее уехать. Теперь мы добавим несколько специальных представлений, которые будут отображать положение машинки на сцене и направление ее движения.

Специальные представления

Чтобы проиллюстрировать, как легко в шаблон МПК могут быть добавлены специальные представления, мы реализуем еще два дополни-

тельных представления. Первое из них будет показывать направление движения машинки, а второе – позицию машинки на сцене, подобно отображению автомобиля на дисплее навигатора GPS.

Представление для показа направления движения

Пример 12.29 показывает нам класс такого представления, DirectionGaugeView. Это оконечное представление, которое использует круговой указатель и стрелку для показа направления движения машинки. Стрелка является спрайтом, чей угол поворота устанавливается таким же, как и у машинки.

Пример 12.29. DirectionGaugeView.as

```
package {
    import flash.geom.*;
    import flash.events.*;
    import flash.display.*;

    public class DirectionGaugeView extends ComponentView
    {

        private var guageHand:Sprite;

        public function DirectionGaugeView(aModel:Object,
                                           aController:Object = null)
        {
            super(aModel, aController);

            // рисуем круг
            graphics.lineStyle(2, (model as ICar).getColor());
            graphics.drawCircle(10, 10, 20);

            // рисуем стрелку
            guageHand = new Sprite();
            guageHand.graphics.lineStyle(2, 0x000000);
            guageHand.graphics.moveTo(0,0);
            guageHand.graphics.lineTo(15,0);
            guageHand.x = guageHand.y = 10;
            this.addChild(guageHand);
        }

        override public function update(event:Event = null):void
        {
            this.guageHand.rotation = (model as ICar).getRotation();
        }
    }
}
```

Представление типа GPS навигатора

Класс GPSView, показанный в примере 12.30, рисует прямоугольник, пропорциональный размерам сцены, и отображает на нем положение машинки относительно границ сцены.

Пример 12.30. GPSView.as

```
package
{
    import flash.geom.*;
    import flash.events.*;
    import flash.display.*;

    public class GPSView extends ComponentView
    {
        private var carPos:Sprite;
        private static const SF:Number = 0.15; // коэффициент масштаба

        public function GPSView(aModel:ICar, target:Stage)
        {
            super(aModel);

            // рисуем прямоугольник пропорционально сцене
            graphics.lineStyle(2, (model as ICar).getColor());
            graphics.drawRect(0, 0, target.stageWidth * SF,
                                                target.stageHeight * SF);

            // рисуем указатель как спрайт
            carPos = new Sprite();
            carPos.graphics.beginFill(0x000000); // черный цвет
            carPos.graphics.drawRect(-2, -2, 5, 5);
            carPos.graphics.endFill();
            this.addChild(carPos);
        }

        override public function update(event:Event = null):void
        {
            var pt:Point = (model as ICar).getLoc();
            this.carPos.x = pt.x * SF;
            this.carPos.y = pt.y * SF;
        }
    }
}
```

Добавление специальных представлений

Добавление новых, специальных представлений в структуру вложенных представлений осуществляется очень просто, поскольку не нужно волноваться о регистрации их обработчиков сообщений об обновлениях. Мы можем просто добавить два композитных представления в каче-

стве дочерних узлов корневого узла в общую структуру представлений. Следующий фрагмент кода нужно вставить на место строки 23 в примере 12.28, чтобы все это реализовать.

```
// представление направления (компонент)
var dash:ComponentView = new DirectionGaugeView(carModel);
kbInputView.add(dash); // добавляем отображение в kbInputView
                       // как дочерний узел
dash.x = dash.y = 20;
addChild(dash);

// GPS-представление (компонент)
var gps:ComponentView = new GPSView(carModel, this.stage);
kbInputView.add(gps); // добавляем представление GPS в kbInputView
                      // как дочерний узел
gps.x = 10;
gps.y = 75;
addChild(gps);
```

Добавление преследующей машинки

А насколько сложно будет добавить в наше приложение вторую машинку, преследующую машинку, движение которой управляется с клавиатуры? Преследующая машинка должна сама поворачиваться, чтобы настигать машинку, управляемую пользователем. Мы используем простой алгоритм для преследователя. Если убегающая машинка находится слева от преследующей, то последняя слегка повернет налево, а если справа, то преследующая машинка повернет направо.

Единственным новым элементом будет новый контроллер для преследующей машинки. В примере 12.31 показан интерфейс такого контроллера `IChaseHandler`, в котором определяется указанный алгоритм преследования, использующий события таймера. Кроме того, в нем определяется метод для указания преследуемой машинки типа `ICar`.

Пример 12.31. IChaseHandler.as

```
package
{
    import flash.events.*;

    public interface IChaseHandler
    {
        function chaseHandler(event:TimerEvent):void
        function setChaseTarget(car:ICar):void
    }
}
```

Класс `ChaseController`, показанный в примере 12.32, имплементирует интерфейс `IChaseHandler`. Он устанавливает таймер на вызов метода об-

работчика chaseHandler() каждую 1/20 долю секунды, чтобы сообщать модели, на сколько нужно изменить угол поворота преследующей машинки. Установить период таймера равным частоте смены кадров для нашего приложения – весьма хорошая идея. Следует обратить внимание также на угол поворота контроллера преследующей машинки. Он составляет только половину от угла поворота машинки, контролируемой пользователем (см. пример 12.25). Преследующая машинка может поворачивать только на 4 градуса вместо 8 (для преследуемой машины). Это должно позволить машинке, контролируемой пользователем, убегать от своего преследователя с помощью нескольких крутых поворотов.

Пример 12.32. ChaseController.as

```
package
{
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.Timer;

    public class ChaseController implements IChaseHandler
    {
        private var model:ICar;
        private var target:ICar;

        public function ChaseController(aModel:ICar)
        {
            this.model = aModel;

            // устанавливаем период вызова контроллера погони равным 1/20 сек.
            var timer:Timer = new Timer(1000 / 20);
            timer.addEventListener(«timer», chaseHandler);
            timer.start();
        }

        public function chaseHandler(event:TimerEvent):void
        {
            var myLoc:Point = model.getLoc();
            var targetLoc:Point = target.getLoc();
            var myRotationAngle:Number = model.getRotation();
            var angleToTarget:Number = Math.atan2(targetLoc.y - myLoc.y,
                                        targetLoc.x - myLoc.x) * 180 / Math.PI;

            if ((myRotationAngle % 360) < angleToTarget)
            {
                model.addToRotationAngle(4);
            } else {
                model.addToRotationAngle(-4);
            }
        }
```

```
        public function setChaseTarget(car:ICar):void
        {
            target = car;
        }
    }
}
```

Данный контроллер реализует некоторый алгоритм преследования. Следовательно, здесь может оказаться полезным использование шаблона Стратегия. Тогда мы могли бы просто заменять наш контроллер на другой, с более хитрым алгоритмом преследования, без каких-либо изменений в других элементах шаблона МПК. Следующий код добавит машинку преследователя в наше приложение.

```
// ** догоняющая машинка **
var chaseCarModel:ICar = new CarModel();
chaseCarModel.setLoc(new Point(100,100));
chaseCarModel.setColor(0xFF0000); // красный

var chaseCarController:IChaseHandler = new ChaseController(chaseCarModel);
chaseCarController.setChaseTarget(ICar(carModel));

// представление догоняющей машинки (компонент)
var chaseCarView:ComponentView = new CarView(chaseCarModel);

addChild(chaseCarView);

// регистрируем машинку на получение обновления модели
chaseCarModel.addEventListener(Event.CHANGE, chaseCarView.update);
```

Обратите внимание, что преследующая машинка состоит из новой триады элементов МПК, отдельной от первой машинки. Она имеет свои модель, представление и контроллер. Мы повторно использовали классы `CarView` и `CarModel`, но изменили поведение преследующей машинки, заменив ей контроллер. Комбинирование новых или подклассов существующих моделей, представлений и контроллеров делает приложения, разработанные с помощью шаблона МПК, просто бесконечно расширяемыми.

Заключение

Шаблон Модель-Представление-Контроллер (МПК) очень широко применяется для создания приложений с пользовательским интерфейсом. Вся мощь данного шаблона определяется разделением ответственности между тремя элементами, которые его составляют. Его *модель* содержит данные приложения и логику управления его состояниями. *Представление* отображает пользовательский интерфейс и состояние приложения на экране. А *контроллер* обрабатывает ввод пользователя, изменяющий состояние приложения.

Шаблон МПК может использовать в своей реализации шаблоны Наблюдатель, Стратегия и Композиция для управления зависимостями внутри и между своими элементами. Взаимодействие между моделью и представлениями может строиться по образцу взаимодействия между *конкретной рассылкой* и *конкретными наблюдателями* в шаблоне Наблюдатель. Взаимодействие между представлением и контроллером – по образцу взаимодействия между классом *контекста* и *конкретной стратегии* в шаблоне Стратегия. Представления в МПК могут содержать в себе многочисленные вложенные представления, взаимодействие между которыми может быть организовано как между *оконечными компонентами* и *композитными узлами* в шаблоне Композиция.

Но еще более важно, что шаблон МПК является образцом для проектирования приложений. Его разделение ответственностей между моделью, представлением и контроллером позволяет легко заменять эти элементы без нарушения работы остальных частей программы. Это дает возможность легко расширять приложения, основанные на шаблоне МПК, и реализовывать в них новые требования.

13

Шаблон Симметричный заместитель

Аллегория – это всего лишь перевод абстрактных понятий на иллюстративный язык, который сам по себе является абстракцией от предметов ощущений; даже более бесполезный, чем его фантомный заместитель, причем они оба одинаково нематериальны, а шаблон бесформенный до своего воплощения. С другой стороны, символ характеризуется тем, что в нем видовое просвечивает сквозь индивидуальное, общее – сквозь видовое, универсальное – сквозь общее. А всего более тем, что вечное просвечивает сквозь временное. Символ всегда причастен реальности, которую делает постижимой; и пока он выражает целое, он остается живой частью того единства, которое он представляет.

Сэмюэл Тэйлор Кольридж

Нет другой такой страны в мире, где машины столь прекрасны, как в Америке. Я понял это, когда смотрел на работу шлюзов в Чикаго; тогда я осознал все великолепие машин. Подъем и опускание стальных стержней, симметричное движение громадных колес – это самое красивое ритмическое зрелище, которое я когда-либо видел.

Оскар Уайльд

Все продающиеся компьютеры симметричны и не ориентированы на какую-либо из рук. Но возможно, что часть существующего программного обеспечения или некоторые алгоритмы являются ориентированными на левшей или правшей.

Филипп Эмегвали

Одновременные ходы и результаты в играх

Одной из самых интересных проблем, возникающих при программировании игр, является взаимодействие игроков через Интернет, особенно в тех играх, в которых разработчики должны предусмотреть для двух или более игроков способы их взаимодействия в контексте установленных правил игры. Данный процесс становится еще более сложным, когда игроки делают свои ходы одновременно (параллельно), и игроки не знают результатов своих ходов, пока они оба их не совершат.

Одной из таких игр является «камень, ножницы, бумага» (КНБ). В ней два игрока одновременно выбрасывают вперед свою ладонь, показывающую либо камень (кулак), либо бумагу (плоская ладонь), либо ножницы (средний и указательный пальцы изображают V-образный знак). Камень побеждает ножницы, ножницы побеждают бумагу, бумага побеждает камень. (См. детали на сайте *http://www.worldrps.com*.) Игра с такими простыми и ясными правилами является идеальным примером для опробования шаблонов проектирования, обеспечивающих параллельное взаимодействие в Интернете.

Концепция шаблона проектирования Симметричный заместитель впервые была представлена в работе «Pattern for Distributing Turn-Based Games» («Шаблон для распределенных, пошаговых игр»), написанной Джеймсом Хелиотисом (James Heliotis) и Акселем Шрейнером (Axel Schreiner), оба из Рочестерского института технологий. Презентация на основе этой работы была сделана Акселем Шрейнером на конференции OOPSLA 2006 г., в Портленде, штат Орегон.

Если вы работаете с шаблонами проектирования, то рано или поздно это приведет вас на OOPSLA. Данное сокращение означает «Object-Oriented Programming, Systems, Languages & Applications» («Объектно-ориентированное программирование, системы, языки и приложения»). OOPSLA – это ежегодная конференция в США, посвященная различным темам ООП, как чисто научным, так и практическим. В 1991 году на конференции OOPSLA была представлена первая часть каталога шаблонов проектирования, с тех пор данная тема занимает прочное место на конференциях OOPSLA. Европейским аналогом такой конференции является ECOOP – «European Conference on Object-Oriented Programming» («Европейская конференция по объектно-ориентированному программированию»).

То, что будет описано в данной книге, основано на вышеуказанных работе и презентации, но пример в данной книге разработан независимо, и мы ни в коей мере не можем возложить ответственность за него на доктора Хелиотиса или доктора Шрейнера. Частично потому, что мы использовали ActionScript 3.0 и Flash Media Server 2, и частично потому, что мы несколько отклонялись в деталях от их инструкций. Тем не менее мы всячески старались сохранить исходные концепции и ар-

гументацию авторов. Благодаря им у нас все получилось правильно, но ответственность за неточности мы берем на себя.

Игрок

- В основе идеи шаблона Симметричный заместитель лежит концепция интерфейса игрока, содержащего методы для игры в игру (*любую*, не только пошаговую), которая может проходить через Интернет. Наш список методов слегка отличается от списка Хелиотиса и Шрейнера, но базовые его концепции взяты нами из их работы.
- Сделать ход (`makeMove`). Каждому игроку нужен метод для передачи ввода пользователя через его интерфейс (UI).
- Отобразить ход локального игрока (`localMove`). После сделанного хода, игра требует, чтобы соответствующее состояние (локальный игрок сделал ход) было записано. Это предупреждает данного игрока от выполнения еще одного хода прежде, чем наступит следующий этап игры.
- Отобразить ход удаленного игрока (`onProxyMove`). Запись хода удаленного игрока и предотвращение его ходов до следующего этапа игры.
- Послать ход локального игрока удаленному игроку (`doMove`). Ход будет передан методу `onProxyMove` оппонента.
- Завершить данный этап игры (`takeTurn`) после того как оба игрока сделали свои ходы. Ходы обоих игроков передаются Судье для определения, кто из них выиграл, и установления значений для следующего этапа.
- Остановить игру до установления связи между обоими игроками (`numConnect`). Этот метод позволяет обоим игрокам узнать о присутствии друг друга и запрещает совершать ход при наличии только одного игрока.

Рассмотрев все составные части этапа игры, мы заложили основу для создания пошаговой игры. Теперь нужно поместить их в контекст игры.

Судья

Чтобы игра шла честно, нам требуется судья. Диаграмма классов для него во взаимоотношении с игроком показана на рис. 13.1.

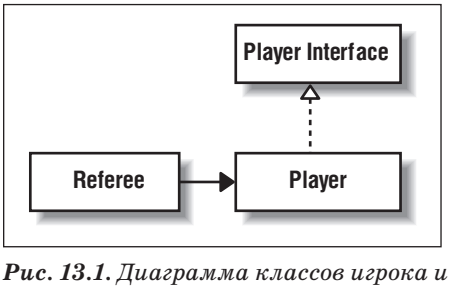


Рис. 13.1. *Диаграмма классов игрока и судьи*

По существу судья (referee) будет классом, содержащим правила игры, заключенные в оболочку его шаблонного метода. Как и в одноименном шаблоне проектирования, методы этого класса должны быть максимально общими, на тот случай, если мы захотим использовать его в другой игре. Используя Шаблонный метод, мы можем зафиксировать порядок вызова операций в методах судьи, но мы хотим сохранить в нем некоторую гибкость для других игр. Так что мы начнем с перечисления операций, касающихся судьи, которые применимы в любых играх:

- Оценить ходы и определить, кто выиграл. В любой игре должен быть набор правил, позволяющий определить победителя и проигравшего или зафиксировать ничью.
- Огласить результаты игры. Как только результаты определены, они должны быть переданы всем участникам игры.
- Сбросить переменные игры в начальное состояние. Поскольку все игроки отслеживают, кто и как походил и кто из игроков выиграл, все переменные, отслеживающие эти состояния, должны очищаться.

Каждая операция оформляется как отдельный метод, а порядок их вызова закрепляется в шаблонном методе. Следуя Голливудскому принципу (см. главу 9), объект судьи сам вызывает объекты игроков.

Теперь, когда у нас есть судья, нам нужно рассмотреть его взаимоотношения с игроками (player 1 и player 2). Рисунок 13.2 показывает диаграмму объектов в игре с двумя игроками и судьей:

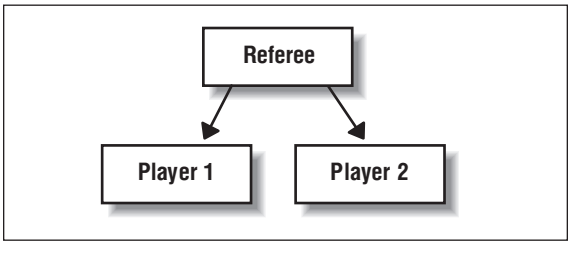


***Рис. 13.2.** Судья и игроки*

Как и в реальной игре, судья принимает решение о ее результатах для двух или более игроков. Пока что наша игра понимается в самом общем виде, она может быть почти любой конкретной игрой. Посмотрев на эту диаграмму объектов, нетрудно вообразить двух или более игроков, взаимодействующих описанным образом на одном компьютере. Если бы нашей задачей было спроектировать игру, в которой один из игроков – компьютер, а другой – пользователь, взаимодействующий с ним с помощью интерфейса пользователя, то мы бы ее уже выполнили. В игре «камень, ножницы, бумага» одного из игроков можно легко заменить компьютерным генератором случайных чисел. Поскольку мы хотим играть через Интернет, нам нужно сделать еще кое-что.

Интернет, игроки и их заместители

Как только у нас появляются удаленные игроки в сети Интернет, все меняется. Каждый игрок делает ход, ход отсылается по сети, судья подсчитывает результат, публикует его и очищает все лишнее. Когда игроки ходят по очереди, задача не выглядит устрашающе, поскольку после каждого хода каждый может видеть ход другого. Судья может подождать, пока не появится первый выигравший, и затем разослать всем результаты. Однако вполне вероятно возникновение затруднения, где именно расположить судью. Можно, например, поместить его на компьютере, с которого начинает игру первый игрок, и общаться тогда со вторым игроком через его заместителя.

Другое решение предполагает размещение судьи на отдельном сервере, в этом случае каждый ход будет проходить через Интернет дважды. И в зависимости от используемого сервера, вероятно, придется несколько переделать проект игры, чтобы приспособить его под возможности судьи.

И хотя оба решения вполне работоспособны, в них мало гибкости и возможностей по повторному использованию кода, необходимых, чтобы считать их полноценными шаблонами проектирования. Нам нужно продолжать наши поиски.

Шаблон Симметричный заместитель

Шаблон Симметричный заместитель является одновременно и открытием, и изобретением. Хелиотис и Шрейнер обнаружили, что можно создать заместителей (proxy) объектов двух игроков, соединенных напрямую через Интернет. Тогда на каждой стороне будет один настоящий игрок и один заместитель другого игрока, и каждый из судей будет иметь одну и ту же информацию. Какой бы ход ни сделал любой из игроков, оба судьи увидят его одинаково. У судьи нет никакой информации, откуда именно пришел к ним ход; он судит вслепую, как отметили Хелиотис и Шрейнер. Судье нужно только оповестить игроков, что игра закончилась, и сообщить им ее результаты. Оба судьи работают абсолютно синхронно, поскольку они реагируют на одинаковую информацию о состоянии игры, не зная, локальная она или от заместителя. В итоге при достижении условий конца игры они абсолютно одинаково информируют об этом игроков, очищают все переменные и подготавливаются к следующей игре.

Рисунок 13.3 показывает нам диаграмму объектов шаблона Симметричный заместитель:

Помня о том, что на рисунке изображена диаграмма объектов, а не диаграмма классов, мы можем увидеть на ней, что в шаблоне Симметричный заместитель имеется, так сказать, заместитель каждого игрока в лагере противника. Судьи рассматривают каждую из сторон Симмет-

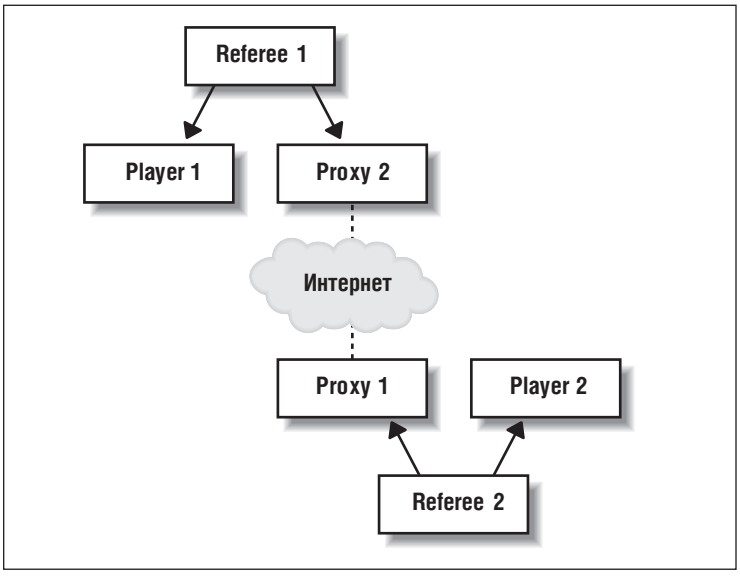


Рис. 13.3. *Диаграмма объектов шаблона Симметричный заместитель*

ричного заместителя как полную игру. Судья просто берет ходы, оценивает их как ходы, выполненные одной и другой стороной, и принимает решения о результатах игры, как будто все происходит на одном компьютере. Другими словами, заместители игроков рассматриваются как компьютеры, играющие за реальных игроков, или как люди, играющие по очереди с одной клавиатуры.

Ключевые особенности

Мы уже описали несколько ключевых особенностей шаблона Симметричный заместитель, но есть еще несколько. В этом разделе мы соберем вместе:

- Интерфейс игрока, включающий в себя свой ход, отображение хода и методы получения результата;
- Класс судьи, основанный на Шаблонном методе и определяющий результаты игры;
- Ячейки синхронизации между пользовательским интерфейсом и игроком;
- Объекты дублированного судьи, игрока и его заместителя.

Мы уже обсудили пользовательский интерфейс и концепцию судьи, нам осталось рассмотреть еще две концепции, прежде чем переходить к вопросу о ключевых концепциях ООП, применимых в данном шаблоне проектирования. Нам нужно, во-первых, познакомиться с ячейками синхронизации и узнать, где они находятся в общей схеме взаимодей-

ствия данного шаблона, а во-вторых, подробнее рассмотреть концепцию объекта заместителя.

Ячейки синхронизации

Хелиотис и Шрейнер называли ячейками в своей схеме контролируемые очереди из одного элемента, размещенные между объектом игрока и графическими элементами. В большинстве случаев графические элементы принадлежали интерфейсу пользователя. При рассмотрении диаграмм последовательностей действий для игр с параллельными ходами игроков такие ячейки помещались между заместителями игроков, по одной на каждого игрока. Таким образом, для одновременных ходов использовались две ячейки; по одной для каждого из игроков, совершающих ход в один и тот же момент времени.

Посмотрев код реализации, использованный Хелиотисом и Шрейнером, мы придумали альтернативную имплементацию, с помощью разделяемых удаленных объектов, доступных во Flash Media Server 2 (FMS2). Такие разделяемые объекты могут работать как механизмы синхронизации, гарантирующие, что обе стороны имеют одну и ту же информацию в каждый момент времени. В случае их использования, как только один из игроков делает свой ход, он сразу же становится доступным объекту его заместителя. То есть «ячейка синхронизации» оказывается внутри игрока в функции, ожидающей ходов обеих сторон. Таким образом, роль ячейки синхронизации берут на себя разделяемые объекты. Передача хода в разделяемый объект ложится на серверный скрипт FMS2.

Заместитель

Поскольку FMS2 берет на себя все проблемы по синхронизации, и он полностью доступен подключенным к нему клиентам через их методы, мы приходим к ситуации, в которой мы должны спросить себя, действительно ли нам нужны отдельные объекты для заместителей игроков? Возможно ли создать не четыре объекта игроков, а только два? Тогда бы объект каждого игрока, основанный на общем интерфейсе, содержал в себе и реального игрока, и его представителя. Это усеченный вариант реализации шаблона Симметричный заместитель, но концептуально он аналогичен исходному шаблону. Рисунок 13.4 показывает диаграмму объектов для такого варианта.

Различия между рис. 13.3 и 13.4 состоят в том, что на последнем заместитель и игрок помещены в один объект и немного больше разъяснены детали передачи информации через Интернет. Такая же модель могла бы использовать для пересылки ходов игроков PHP, ColdFusion или любой другой механизм пересылки информации через Интернет. (Аксель Шрейнер отмечал, что можно применять даже дымовые сигналы, поскольку способ пересылки информации по сети для нас совершенно не важен.)

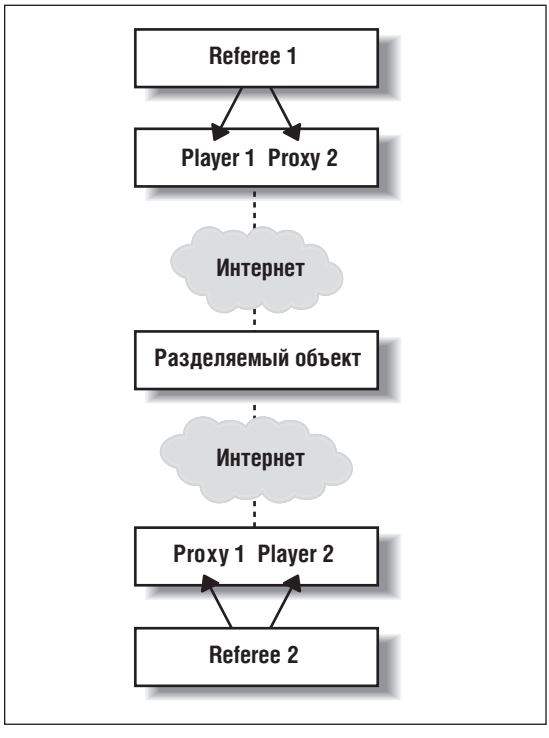


Рис. 13.4. *Усеченный вариант Симметричного заместителя*

Но есть один момент, который нам не очень нравится в такой усеченной реализации Симметричного заместителя, – некоторая потеря гранулярности его исходной модели. Поместив представителя в тот же объект, что и игрока, мы их соединили вместе, что может затруднить адаптацию такой схемы для других игр. Однако сейчас никаких подобных затруднений не наблюдается. Оба игрока совершенно симметричны, и роль судьи в данном варианте полностью сохранена.

Ключевые концепции ООП в шаблоне Симметричный заместитель

Все концепции ООП, которые обсуждались нами для различных шаблонов проектирования, входящих в состав составных шаблонов, применимы и здесь. Чтобы избежать повторения рассмотрения ключевых концепций ООП, проведенного нами в главах, посвященных отдельным шаблонам проектирования, мы обратим свое внимание на более общую концепцию ООП для шаблона Симметричный заместитель – *гибкость*.

Поскольку Симметричный заместитель специализируется на взаимодействии через Интернет, нам нужно познакомиться с двумя разными

видами его гибкости. Во-первых, нам необходимо рассмотреть гибкость коммуникаций, то есть может ли наша модель быть использована с различными технологиями коммуникации через Интернет. Во-вторых, нам нужно рассмотреть гибкость относительно различных игр, то есть насколько широко различные игры могут использовать подобную схему в своей работе.

Коммуникационная гибкость

Первым делом нам нужно выяснить, может ли шаблон Симметричный заместитель использовать любую систему коммуникации через Интернет. Использование таких приложений, как FMS2, применяющих свои встроенные возможности, вроде удаленных разделяемых объектов, не очень удобно для тестирования коммуникационной гибкости и как платформа разработки для связи внутри шаблона. Специальный скрипт на стороне сервера автоматически информирует всех игроков о текущем состоянии игры и состоянии ходов всех игроков. Как только оба игрока сделали свои ходы, судья определяет, кто из них победил, и производит инициализацию игры для следующего раунда.

В исходной модели Симметричного заместителя и в ее усеченном варианте, пока заместитель игрока делает те же ходы, что и сам игрок, не имеет особого значения, применяется ли технология FMS, другая технология на основе открытых сокетов либо такие технологии, как PHP, C#, Perl, ColdFusion и VB.NET. Единственное состояние, которое действительно нужно переслать через Интернет, – это ход для заместителя игрока. Заместитель делает ход, судья определяет, кто выиграл, публикует результаты и переустанавливает все переменные для нового раунда игры. Поэтому любая технология коммуникации, которая может переслать состояние игрока к его заместителю, приемлема для данного шаблона, что говорит о его коммуникационной гибкости.

Игровая гибкость

В имплементации игры, которую вы увидите в данной главе, судья не так активен, как предполагалось в работе Хелиотиса и Шрейнера. Основное его отличие состоит в том, что он вмешивается в игру, только когда оба игрока закончили свои ходы. Поскольку в игре КНБ каждый раунд является полной игрой, она заканчивается сразу после того как оба игрока сделали ходы. В этот момент судья запускает свой шаблонный метод, определяет победителя, публикует результаты и переустанавливает все переменные в начальные состояния.

Одним из способов сокращения работы судьи является перенесение части функций в интерфейс пользователя. Игрок может сделать выбор только из трех вариантов кнопок, которые у него есть. Объект реального игрока блокирует попытки сделать какие-либо другие ходы, включая и такую, при которой игрок вообще ничего не выбирает. В более сложных играх, где у игроков много различных вариантов хода, на судью

или интерфейс пользователя может возлагаться существенно больше ответственности по ограничению набора возможных ходов в зависимости от ситуации игры.

Имея шаблонный метод и три входящих в него метода в своем абстрактном классе, класс судьи `Referee` обладает и гибкостью, и функциональностью. В нашем учебном приложении в данной главе у класса `Referee` будет подкласс `RPS` («КНБ»), который будет описывать детали, специфические для игры КНБ. В данном случае мера игровой гибкости состоит в том, смогут ли различные игры реализовываться с помощью подклассов класса `Referee`.

Из трех функций, составляющих шаблонный метод, гибкость наиболее важна для функции, определяющей победителя в игре или ничью. Вот как эта функция определена в классе `Referee`:

```
function doWinner(p1Move:String,p2Move:String):String
{
    return winner;
}
```

Поскольку данный метод находится в абстрактном классе, мы предполагаем, что она будет замещена при реализации различных игр. Она принимает в качестве параметров ходы игроков 1 и 2, не имея понятия о том, пришли они от заместителей или нет, и возвращает строку с победителем или ничьей. Будучи буквально абстрактной, она просто ждет, когда один их двух ходов выполнит условия выигрыша или ничьей. Для игры КНБ все это обстоит относительно просто, поскольку в ней каждый раунд составляет полную игру.

А как насчет игры типа крестики-нолики? Эта игра состоит из нескольких различных раундов и обладает более широким спектром результатов. Поскольку наш метод `doWinner()` включает в себя параметры от обоих игроков, он может обработать все ходы. Каждый ход может быть описан в терминах матрицы 3 на 3 с адресами ячеек от C1R1 до C3R3 (номер колонки/номер строки). Когда один из игроков выполнит условия победы, или игра достигнет точки, где никто из них не может выиграть, тогда может быть запущен полный шаблонный метод.

Возможен и другой вариант: каждый ход передается классу `Referee` либо его подклассам. Именно так и хотели сделать Хелиотис и Шрейнер, и гибкость шаблона проектирования Симметричный заместитель позволяет использовать любую из указанных альтернатив. В играх, в которых участвует более двух игроков, команды игроков или комбинации игроков и команд, достаточно всего нескольких изменений в параметрах метода `doWinner()`, чтобы он мог работать и в более сложных ситуациях.

Ключ к игровой гибкости шаблона Симметричный заместитель лежит в возможности замещать методы в исходном шаблонном методе класса

Referee. Таким образом, создание многопользовательской игры на основе игры для двух участников довольно легко реализуется, пока у каждого из игроков есть «свой» судья и все ходы немедленно пересылаются всем игрокам и/или судьям, и не требует никаких изменений в общей схеме шаблона проектирования.

Интерфейс игрока

Интерфейс игрока является отправной точкой при реализации шаблона проектирования Симметричный заместитель. Данный интерфейс содержит шесть методов. Как и все интерфейсы, он содержит абстрактные методы и их параметры. Пример 13.1 показывает интерфейс ISymPlayer.

Пример 13.1. ISymPlayer.as

```
package
{
    // Интерфейс симметричного заместителя
    import flash.events.Event;

    interface ISymPlayer
    {
        function numConnect(cl:uint):void;
        function makeMove(event:Event):void;
        function doMove(s:String):void;
        function localMove(locMove:String):void;
        function onProxyMove(proxMove:String):void;
        function takeTurn():void;
    }
}
```

Здесь мы представим вам краткий обзор того, что делает каждый из этих методов. Для простоты восприятия все эти функции расположены в приблизительном порядке их вызова, за исключением первой, которая выполняет рутинную работу проверки того, что оба игрока подключены к FMS, до того, как любой из них сможет сделать ход.

1. Прежде всего метод numConnect() проверяет, сколько клиентов подключено, и, если подключены оба, позволяет сделать ходы обоим игрокам.
2. Игроки выбирают свои варианты хода путем нажатия на одну из трех возможных кнопок (камень, ножницы или бумага), которое запускает метод makeMove().
3. Функция makeMove() запускает методы doMove() и localMove().
4. Операция doMove() вызывает сервер, чтобы он передал ход заместителю игрока.

5. Метод `localMove()` сначала сохраняет ход в локальных переменных и затем устанавливает логическую переменную, показывающую, что ход сделан.
6. Далее функция `onProxyMove()` отвечает на вызов со стороны сервера для разделяемых объектов и действует аналогично методу `localMove()`, но по отношению к заместителю игрока.
7. После того как либо `localMove()`, либо `onProxyMove()` показывает, что оба игрока сделали свои ходы, метод `takeTurn()` заставляет `Referee` сделать вызов для определения победителя и переустановки переменных для новой игры.

Различные вариации в имплементации указанных методов определяют, могут ли ходы игроков быть параллельными или только следовать один за другим.

Судья

Поскольку наш класс судьи является абстрактным, он относительно небольшой. Его методы довольно общие и могут быть замещены, однако созданы для определенных целей. Кроме того, поскольку они находятся в шаблонном методе, они должны разрабатываться с учетом возможного порядка их вызова. Давайте сначала взглянем на класс `Referee` целиком (пример 13.2), а затем подробнее рассмотрим его методы.

Пример 13.2. Referee.as

```
package
{
    // Абстрактный класс
    public class Referee
    {
        // Ходы
        private var p1Move:String;// ход 1-го игрока
        private var p2Move:String;// ход 2-го игрока
        private var winner:String;// значение для победителя
        private var outcome:String;// описание победителя
        private var displayWindow:DynamicText;
        private var movecheck:Array;// массив истории ходов

        // Шаблонный метод
        final function moveComplete(p1Move:String,p2Move:String,
            displayWindow:DynamicText,movecheck:Array):void
        {
            outcome=doWinner(p1Move,p2Move);
            displayResults(displayWindow,outcome);
            resetGame(movecheck);
        }
```

```
        // Абстрактные методы

        protected function doWinner(p1Move:String,p2Move:String):String
        {
            return winner;
        }
        function displayResults(displayWindow:DynamicText,
            outcome:String):void
        {
            displayWindow.setMove(outcome);
        }
        protected function resetGame(movecheck:Array):void
        {
            for (var r:uint =0; r< movecheck.length; r++)
            {
                movecheck[r]=false;
            }
        }
    }
}
```

Комментарий в самом начале класса о том, что он является абстрактным, – это просто комментарий. Он лишь напоминание, что в ActionScript 3.0 нет настоящих абстрактных классов, и мы должны сами помнить, что нужно использовать замещение методов. Комментарий (`// Абстрактные методы`), показывающий абстрактные методы, также служит напоминанием, что это не настоящие абстрактные методы, поскольку они не поддерживаются в ActionScript 3.0.

Методы

Первый из трех методов класса `Referee` должен быть очень гибким, поскольку он определяет победителя в игре. Его реальная имплементация будет находиться в подклассах, описывающих игру. Данный метод `doWinner()` скорее всего будет находиться в том из них, где определяются правила игры.

```
function doWinner(p1Move:String,p2Move:String):String
{
    return winner;
}
```

Он включает в себя два параметра для хода каждого из игроков. В контексте игры КНБ, где в каждой игре происходит только один ход для каждого из игроков, эти ходы определяют конечное состояние игры. Однако в других играх класс `Referee`, возможно, придется вызывать на каждом ходе, чтобы он мог накопить в себе информацию, необходимую для определения победителя.

Следующий метод предназначен для публикации результатов и получает для этого в качестве своих параметров ссылки на текстовое поле и строку.

```
function displayResults(displayWindow:DynamicText,outcome:String):void
{
    displayWindow.setMove(outcome);
}
```

В контексте игры КНБ, да и большинства других игр, этот метод имеет две задачи. Как часть шаблонного метода он показывает, кто из игроков выиграл. Однако он может быть также использован для показа другой информации, не зависящей от шаблонного метода. Вспомните, что ходы в КНБ происходят одновременно, и ни один из игроков не может видеть ход другого игрока, пока оба они не походят. Значит, данный метод можно использовать для показа другой информации, такой как ход противника, всякий подходящий для этого раз.

Третий метод в классе `Referee` предназначается для установки всех значений переменных в исходное состояние, то есть возврата всех фигурок в шахматах на исходные позиции, так сказать. Этот метод носит обслуживающий характер, но он важен, если вы хотите играть в игру несколько раз без ее перезагрузки.

```
function resetGame(movecheck:Array):void
{
    for (var r:uint =0; r< movecheck.length; r++)
    {
        movecheck[r]=false;
    }
}
```

Если вы храните все ходы игры в одном массиве, то восстановление исходного состояния игры происходит легче и более гибко. Когда ходов всего два, эта функция могла бы просто устанавливать две логические переменные в значение ложь. (Такое их состояние означало бы, что ходов *не было*.) Поскольку она работает с массивом, ей не важно, сколько вообще ходов было сделано игроками. То есть в таком виде она более гибкая и универсальная, чем с использованием отдельных переменных.

Шаблонный метод

Последний метод класса `Referee` создается на основе уже перечисленных его методов. Поскольку это шаблонный метод, он фиксируется с помощью ключевого слова `final`. Как вы помните, ключевое слово `final` запрещает замещение данного метода в подклассах. Однако методы, которые *входят* в шаблонный метод, сами могут замещаться, и мы ожидаем, что по крайней мере для некоторых из них так оно и будет.

```
final function moveComplete(p1Move,p2Move,displayWindow,movecheck):void
{
```

```
        outcome=doWinner(p1Move,p2Move);
        displayResults(displayWindow,outcome);
        resetGame(movecheck);
    }
```

В данном конкретном шаблонном методе первая операция передает результаты игры, строковую переменную. Потом эта переменная используется во втором методе для показа результатов игры с помощью специализированных объектов. В конце шаблонный метод возвращает игру в ее исходное состояние. Проще говоря, шаблонный метод делает следующее:

1. Определяет победителя или ничью и помещает эту информацию в переменную.
2. Публикует результаты игры.
3. Возвращает игру в начальное состояние.

Таким образом, при использовании класса `Referee` вся информация об игре аккуратно упакована и подготовлена для определения победителя, публикации результатов и возвращения к началу игры.

Подкласс КНБ

Класс `Referee` предназначен только для создания подклассов, а его методы – для замещения, чтобы разработчики могли использовать этот шаблон проектирования для игр различных типов. В примере 13.3 показан его подкласс `RPS` («КНБ»), разработанный для игры «камень, ножницы, бумага».

Пример 13.3. RPS.as

```
package
{
    // Камень, Ножницы, Бумага
    public class RPS extends Referee
    {
        private var winner:uint;
        private var gameOver:Array;
        private var winNow:String;
        //
        override protected function doWinner(p1Move:String,
            p2Move:String):String
        {
            if (p1Move=="rock")
            {
                switch (p2Move)
                {
                    case “rock” :
                        winner=2;
                        break;
```

```
                    case “paper” :
                        winner=1;
                        break;

                    default :
                        winner=0;
                }
            }
            else if (p1Move==”paper”)
            {
                switch (p2Move)
                {
                    case “rock” :
                        winner=0;
                        break;

                    case “paper” :
                        winner=2;
                        break;

                    default :
                        winner=1;
                }
            }
            else
            {
                switch (p2Move)
                {
                    case “rock” :
                        winner=1;
                        break;

                    case “paper” :
                        winner=0;
                        break;

                    default :
                        winner=2;
                }
            }
            gameOver=new Array(“p1 Wins!”,”p2 Wins!”,”Tie!”);
            winNow=gameOver[winner];
            return winNow;
        }
    }
}
```

В данном классе `RPS` **замещается метод** `doWinner()`**, объявленный в родительском абстрактном классе** `Referee`**. Остальные два метода из этого класса используются без всяких изменений. Простой алгоритм опреде-**

ляет, какая из сторон победила в игре или произошла ли ничья, потом переносит эту информацию в элемент массива, сохраняет в строковой переменной и возвращает ее.

Если вы спросите, не проще ли было просто сделать `winner` строковой переменной и вернуть ее, то ответ будет положительный. Однако нам понравилась идея завести один массив, куда вы могли бы поместить ваши собственные выражения или «крепкие словечки» для показа результатов. Например, вместо простого «p1 Wins!» («Выиграл p1!») вы могли бы написать что-то вроде «The Mighty Player 1 Conquers All!» («Великий игрок номер один Победил Всех!») или что-то более креативное.

При использовании нашего класса `RPS` мы будем следовать рекомендации применять интерфейсы, а не их имплементации. Поэтому при объявлении объектов, которые будут использовать данный класс, мы будем объявлять их через класс `Referee`, а не `RPS`. (Посмотрите на создание объектов `RPS` в классах `SymPlayer1` и `SymPlayer2`.)

Информация, разделяемая через Интернет

Сейчас нам нужно немного отойти в сторону и обсудить технологии передачи данных через Интернет, которые мы используем для работы наших заместителей игроков. Как мы упоминали ранее в этой главе, мы решили применять Flash Media Server 2 для передачи информации о ходе игрока его заместителю на другой стороне. Чтобы подробно изучить такое взаимодействие, мы рассмотрим серверный сценарий для FMS2 и те части классов игрока и заместителя, которые его используют.

В примере 13.4 показан файл серверного сценария *proxygame.asc* (все буквы строчные). Он написан на ActionScript 1.0, поскольку во времена его написания все серверные сценарии могли быть написаны только на варианте языка Server Side Communication ActionScript (SSCA). Но все его клиенты написаны на ActionScript 3.0. Данный сценарий должен находиться на сервере FMS2. Часто *swf*-файлы, содержащие скомпилированные приложения и серверные сценарии в *asc*-файлах, помещают на том же компьютере, но не всегда. Взаимодействие с сервером осуществляется через RTMP-протокол, который входит в клиентский сценарий.

Пример 13.4. proxygame.asc

```
application.onAppStart=function()
{
    trace(this.name + " is reloaded");
    this.ss_so = SharedObject.get("proxmove",false);
    this.dup1=false;
    this.dup2=false;
```

```
};
//------------------------
application.onConnect=function(currentClient,username)
{

    currentClient.name=username;
    if(currentClient.moveNow==null)
    {
        currentClient.moveNow="ready";
    }

    var cl=application.clients.length;

    // Проверить имя пользователя и узнать, нет ли
        дубликатов или более двух игроков
    if(((username == «player1» && !this.dup1) || (
        username == "player2" && !this.dup2)) && cl<=1)
    {
     this.acceptConnection(currentClient);
     this.ss_so.setProperty(currentClient.name,username);
        if(username=="player1")
        {
            this.dup1=true;
        }
        else
        {
            this.dup2=true;
        }

    }
    else
    {
     this.rejectConnection(currentClient);
     trace("Connection rejected");
    }

    currentClient.makeMove=function(moveit)
    {
        application.ss_so.setProperty(currentClient.moveNow,moveit)
        someMove=application.ss_so.getProperty(currentClient.moveNow);
        playerNow=application.ss_so.getProperty(currentClient.name);
        someMove+="~"+playerNow;
        application.ss_so.send("onProxyMove",someMove);
    };

    currentClient.checkPlayNum=function()
    {
        var c2=application.clients.length;
        application.ss_so.send("numConnect",c2);
```

```
    }
};

//------------------------
application.onDisconnect = function(currentClient)
{
    dupeName=currentClient.name;
    if(dupeName=="player1")
        {
            this.dup1=false;
        }
        else
        {
            this.dup2=false;
        }
    trace("disconnect: "+currentClient.name);
    this.ss_so.setProperty(currentClient.name,null);
    c2=application.clients.length;
    application.ss_so.send("numConnect",c2);
};
```

Данная программа поделена на три основные части. Первая часть запускается только при старте приложения (`application.onAppStart`). Каждый из игроков может инициализировать сценарий на стороне сервера, и, будучи один раз запущенным, он не может быть запущен, пока оба игрока не закрыли приложение и затем еще в течение приблизительно 20 минут после прекращения работы последнего игрока.

Вам могло показаться, что наше замечание, что версия ActionScript именно 1.0, является просто опечаткой, поскольку все остальные сценарии в данной книге написаны на ActionScript 3.0. Но это не опечатка. Подмножество языка ActionScript для FMS2 очень мало изменилось со времен исходного SSCA, выпущенного для Flash Communication Server (FCS). Когда был выпущен FMS2, ActionScript 2.0 уже входил в комплект Flash, но не мог использоваться для кодирования серверных сценариев. Поэтому SSCA не только отличается от стандартного ActionScript, но и имеет совсем другую версию.

Вторая часть данного сценария запускается во время работы с приложением, когда любой из игроков пытается с ним соединиться (`application.onConnect`). Она генерирует значение по умолчанию «ready» («готово») для переменной `moveNow` и проверяет правильность имен и количество подключенных игроков. Функция, присваивающая ход разделяемому объекту (`currentClient.makeMove`), служит для записи самого хода и того, кто из игроков его сделал. Полученная строка, разделенная символом тильда (~), посылается всем игрокам (`application.ss_so.send`). Символ тильда используется для отделения в строке имени игрока от

его хода. Клиентский сценарий проверяет имя игрока, делающего ход, и присваивает его заместителю. Остальные функции занимаются обслуживанием соединения.

Третья часть серверного сценария (`application.onDisconnect`) выполняет служебные функции, сбрасывая логические переменные. Она также позволяет игроку узнать количество всех игроков, подключенных в текущий момент. При работе с приложениями через Интернет возможность узнать, подключен ли кто-то сейчас к серверу, просто необходима.

Классы игрока и заместителя

Основная имплементация интерфейса симметричного игрока `ISymPlayer` происходит в двух классах, по одному для каждого игрока. Эти классы практически одинаковы, за исключением их внутренних идентификаторов «`player1`» и «`player2`» для сервера. Каждый из них содержит и заместителя для своего оппонента, как уже описывалось в данной главе.

Такой класс показан в примере 13.5. Его можно условно разделить на шесть частей:

1. Импорт необходимых имен и классов.
2. Создание переменных для различных методов.
3. Функция конструктора класса. Она включает в себя все необходимое для установления соединения, вызовы к функциям текстового поля и кнопок и установку обработчиков событий для кнопок.
4. Имплементации шести методов интерфейса в порядке их объявления в нем. Это основная часть данного класса.
5. Проверка установления соединения и, в случае его установления, инициализация разделяемого объекта на стороне клиента. Кроме того, соответствующая функция соединяет разделяемые объекты через объект `nc` – представителя `NetConnection`. Наконец, она вызывает функцию серверного сценария проверки числа игроков `checkPlayNum()`. Этот вызов включает серверную функцию, которая посылает число подключенных игроков назад клиентской функции `numConnect()`.
6. Создание текстовых полей и кнопок.

Рассмотрев совместно примеры 13.4 и 13.5, вы сможете лучше понять схему взаимодействия между классом на стороне клиента и приложением на стороне сервера. (Хотя *asc*-файлы не являются настоящими классами, у них много общего с ними.) Всякий раз, когда вы будете видеть вызов вида `nc.call("functionName",p1,p2)` в классе `SymPlayer1`, он будет означать вызов функции на стороне сервера. В серверном сценарии любой вызов вида `application.ss_so.send("functionName",p1)` означает вызов функции в классах `SymPlayer1` или `SymPlayer2`.

Пример 13.5. SymPlayer1.as

```
package
{
    // Симметричный класс Игрок 1/Заместитель 2

    import flash.net.NetConnection;
    import flash.net.ObjectEncoding;
    import flash.display.Sprite;
    import flash.display.MovieClip;
    import flash.events.Event;
    import flash.events.NetStatusEvent;
    import flash.net.SharedObject;
    import flash.events.MouseEvent;

    public class SymPlayer1 extends Sprite implements ISymPlayer
    {
        private var nc:NetConnection;
        private var rtmpNow:String;
        private var playerNow:String;
        private var cs_so:SharedObject;
        private var playerText:DynamicText;
        private var showText:DynamicText;
        private var oppText:DynamicText;
        private var moveText:DynamicText;
        private var rockBtn:MoveButton;
        private var paperBtn:MoveButton;
        private var scissorsBtn:MoveButton;
        private var moveBtn:MoveButton;
        private var connect:Connect;
        private var moveVal:String;
        private var p1move:String="ready";
        private var p2move:String="ready";
        private var rps:Referee;
        private var winner:uint;
        private var monitor:Array=new Array(false,false,false);
        private var cl:uint;
        private var mcheck:Boolean=false;
        private var connected:String;

        public function SymPlayer1()
        {
            NetConnection.defaultObjectEncoding =
                flash.net.ObjectEncoding.AMF0;
            SharedObject.defaultObjectEncoding =
                flash.net.ObjectEncoding.AMF0;
            setDynamic();
            setButton();
            rockBtn.addEventListener(MouseEvent.CLICK, makeMove);
            paperBtn.addEventListener(MouseEvent.CLICK, makeMove);
```

```
        scissorsBtn.addEventListener(MouseEvent.CLICK, makeMove);
        moveBtn.addEventListener(MouseEvent.CLICK, makeMove);
        //rtmpNow="rtmp://192.168.0.11/proxygame/";
        rtmpNow="rtmp://mojo.iit.hartford.edu/proxygame/";
        //rtmpNow="rtmp:/proxygame/";
        nc = new NetConnection();
        nc.connect(rtmpNow,"player1");
        nc.addEventListener(NetStatusEvent.NET_STATUS,
            checkHookupStatus);
    }
    //
    // Сделать ход
    //
    public function makeMove(event:Event):void
    {
        moveVal=event.currentTarget.name;
        if (!monitor[0] && mcheck)
        {
            switch (moveVal)
            {
                case "scissors" :
                    showText.setMove("scissors");
                    moveText.setMove("scissors");
                    break;

                case "rock" :
                    showText.setMove("rock");
                    moveText.setMove("rock");
                    break;

                case "paper" :
                    showText.setMove("paper");
                    moveText.setMove("paper");
                    break;

                case "move" :
                    var m:String=showText.getMove();
                    if (m != "ready" && m != "Error!")
                    {
                        // Ход Заместителя 1
                        doMove(m);
                        // Ход Игрока 1
                        localMove(m);
                    }
                    else
                    {
                        showText.setMove("Error!");
                    }
            }
        }
    }
```

```
//
// Игрок походил
//
public function localMove(locMove:String):void
{
    playerText.setMove("player1");
    if (!monitor[0])
    {
        p1move=locMove;
        monitor[0]=true;
    }
    // Проверка, походили ли оба игрока
    monitor[2]=(monitor[0] && monitor[1]);
    if (monitor[2])
    {
        takeTurn();
    }
}
//
// Информация с сервера
//
public function onProxyMove(proxMove:String):void
{
    playerNow=proxMove.substring(proxMove.indexOf("~")+1);
    proxMove=proxMove.substring(0, proxMove.indexOf("~"));
    if (playerNow=="player2" && !monitor[1])
    {
        playerText.setMove(playerNow);
        p2move=proxMove;
        monitor[1]=true;
    }
    // Проверка, походили ли оба игрока
    monitor[2]=(monitor[0] && monitor[1]);
    if (monitor[2])
    {
        takeTurn();
    }
}
//
// Вызвать сервер
//
public function doMove(m:String):void
{
    nc.call("makeMove",null,m);
}
//
// Закончить ход
//
public function takeTurn():void
{
```

```
        rps=new RPS();
        rps.moveComplete(p1move,p2move,playerText,monitor);
        rps.displayResults(oppText,p2move);
        showText.setMove("ready");
    }
    // Получить число соединений
    public function numConnect(cl:uint):void
    {
        if (cl==2)
        {
            mcheck=true;
        }
        else
        {
            mcheck=false;
        }
        connected=String(cl+ " connected");
        playerText.setMove(connected);
    }
    //
    // Проверка соединения и создание разделяемых объектов
    //
    public function checkHookupStatus(event:NetStatusEvent):void
    {
        if (event.info.code == "NetConnection.Connect.Success")
        {
            connect.gotoAndStop(2);
            cs_so=SharedObject.getRemote("proxmove",nc.uri,false);
            cs_so.client=this;
            cs_so.connect(nc);
            nc.call("checkPlayNum",null,null);
        }
    }
    // Текст
    public function setDynamic():void
    {
        playerText=new DynamicText();
        addChild(playerText);
        playerText.x=180;
        playerText.y=50;

        showText=new DynamicText();
        addChild(showText);
        showText.x=260;
        showText.y=50;
        showText.setMove("ready");

        moveText=new DynamicText();
        addChild(moveText);
```

```
            moveText.x=180;
            moveText.y=140;

            oppText=new DynamicText();
            addChild(oppText);
            oppText.x=100;
            oppText.y=140;
            oppText.setMove("opponent");
        }
        // Кнопка
        public function setButton():void
        {
            moveBtn=new MoveButton("Make Move",0xcccccc);
            moveBtn.name="move";
            addChild(moveBtn);
            moveBtn.x=100;
            moveBtn.y=50;

            rockBtn=new MoveButton("Rock",0xcccccc);
            rockBtn.name="rock";
            addChild(rockBtn);
            rockBtn.x=100;
            rockBtn.y=80;

            paperBtn=new MoveButton("Paper",0xcccccc);
            paperBtn.name="paper";
            addChild(paperBtn);
            paperBtn.x=150;
            paperBtn.y=80;

            scissorsBtn=new MoveButton("Scissors",0xcccccc);
            scissorsBtn.name="scissors";
            addChild(scissorsBtn);
            scissorsBtn.x=200;
            scissorsBtn.y=80;

            connect=new Connect();
            addChild(connect);
            connect.x=175;
            connect.y=250;
        }
    }
}
```

Здесь все текстовые поля и кнопки являются пользовательскими классами, и они должны быть созданы до начала тестирования данного класса. Далее в этой главе в разделе «Служебные классы и файл документа» вы найдете все необходимые дополнительные классы, включая кнопки и динамические текстовые поля.

Процесс игры

Хотя класс игрока может показаться очень длинным и громоздким, его основные элементы весьма просты. Все в нем сосредоточено на совершении одного из трех вариантов хода – камень, ножницы или бумага. И каждый ход представляет собой просто строковое значение.

Событие хода

Первый шаг инкапсулирован в функции `makeMove()`, которая запускается событием нажатия на кнопку. Используя имя кнопки `name` (а не ее название `label`), она определяет, какой из трех вариантов хода был выбран и нажата ли кнопка `Move`. Для простоты все имена кнопок написаны с помощью маленьких букв, а их названия начинаются с больших букв (например, название Rock (Камень), а имя rock (камень)). Кнопка `Move` позволяет пользователю передумать после выбора одного из трех вариантов хода. Нажав на кнопку `Move`, игрок подтверждает свой выбранный ход и запускает два метода, `doMove()` и `localMove()`.

Двойной ход

Функция, которая делает ход после нажатия кнопки `Move`, посылает его заместителю с помощью метода `doMove()` и сохраняет его локально с помощью `localMove()`. В этом состоит вся суть шаблона Симметричный заместитель. Его симметрия основана на том, что ход посылается как локальному игроку, так и его удаленному заместителю.

Функция локального хода проверяет, что локальным игроком еще не было совершено хода, а затем смотрит, не походил ли другой игрок. Если он походил, то информация о ходах передается судье `Referee` для определения результата игры и его публикации.

А функция `doMove()` по сути является не более чем вызовом сценария на стороне сервера. Она просто передает информацию о ходе игрока серверу и на этом заканчивает свою работу.

Ход заместителя игрока

Метод `onProxyMove()` является зеркальным отражением локального хода `localMove()`. Поскольку данный метод запускается независимо от того, кто из игроков походил, он первым делом отфильтровывает локальные ходы, разделяя строку, переданную от сервера на имя игрока и его ход. Она сравнивает имя игрока с его локальным именем и, если они совпадают, игнорирует данный ход. (Такой ход обрабатывается методом `localMove()`.)

Однако после прохождения фильтрации алгоритм продолжается, точно как при локальном ходе. Он отображает только факт, что удаленный игрок сделал свой ход, но *не* сам этот ход. Таким образом, приложение поддерживает одновременный характер ходов в данной игре.

Объект судьи

Проверка конечного состояния игры делегируется объекту судьи. В данном случае им является представитель подкласса RPS, содержащий метод takeTurn(). Благодаря передаче ходов, текстового поля и имени массива для хранения ходов объекту RPS и его шаблонному методу moveComplete() вся информация отправляется по назначению. Единственная строка

```
rps.moveComplete(p1move,p2move,playerText,monitor);
```

использует преимущества шаблонного метода в классе Referee и запускает составляющие его операции: doWinner(), displayResults() и resetGame().

В дополнение к публикации классом Referee победителя и очистке игры к новому раунду метод displayResults() используется для показа каждому из игроков хода другого игрока. То есть помимо шаблонного метода он применяет один из методов, который в него входит, еще и отдельно.

Изменения во втором игроке

Класс второго игрока также имплементирует собой интерфейс ISymPlayer, но немного по-другому, изменив несколько значений переменных и параметров. Для его реализации вам надо сделать только несколько изменений в классе SymPlayer1. Сначала сохраните файл *SymPlayer1.as* как *SymPlayer2.as*, а потом сделайте в нем изменения, выделенные жирным шрифтом в следующих сегментах кода:

```
    public class SymPlayer2 extends Sprite implements ISymPlayer
....
    public function SymPlayer2()
....
        //
        // Сделать ход
        //
        public function makeMove(event:Event):void
        ....
            moveVal=event.currentTarget.name;
            if (!monitor[1] && mcheck)
            ....
                        var m:String=showText.getMove();
                        if (m != «ready» && m != "Error!")
                        {
                            // Ход заместителя 2
                            doMove(m);
                            // Ход игрока 2
                            localMove(m);
                        }
                        ....
```

```
        //
        // Игрок походил
        //
        public function localMove(locMove:String):void
        {
            playerText.setMove("player2");
            if (!monitor[1])
            {
                p2move=locMove;
                monitor[1]=true;
            }
....
        //
        // Информация с сервера
        //
        public function onProxyMove(proxMove:String):void
        {
            playerNow=proxMove.substring(proxMove.indexOf("~")+1);
            proxMove=proxMove.substring(0, proxMove.indexOf(«~»));
            if (playerNow==»player1» && !monitor[0])
            {
                playerText.setMove(playerNow);
                p1move=proxMove;
                monitor[0]=true;
....
public function takeTurn():void
        {
            rps=new RPS();
            rps.moveComplete(p1move,p2move,playerText,monitor);
            rps.displayResults(oppText,p2move);
            showText.setMove(«ready»);
        }
....
```

То есть просто поменяйте несколько единичек на двойки и несколько нулей на единички в ключевых методах класса `SymPlayer1`, и все будет готово.

Служебные классы и файл документа

Как бы ни были велики классы имплементации игроков, они были бы гораздо больше, если бы часть работы не была сделана вспомогательными классами, создающими кнопки и динамические текстовые поля.

Динамические текстовые поля

Поскольку весь ввод данных в приложение происходит с помощью кнопок, все его текстовые поля для отображения внутренней информации должны быть динамическими. Пример 13.6 демонстрирует такое текстовое поле.

Пример 13.6. DynamicText.as

```
package
{
    import flash.text.TextField;
    import flash.text.TextFieldType;
    import flash.text.TextFormat;
    import flash.display.Sprite;

    public class DynamicText extends Sprite
    {
        private var gameInfo:TextField;
        private var gameFormat:TextFormat;

        public function DynamicText():void
        {
            gameInfo=new TextField();
            addChild(gameInfo);
            gameInfo.border=false;
            gameInfo.background=true;
            gameInfo.type=TextFieldType.DYNAMIC;
            gameInfo.width=70;
            gameInfo.height=20;
            gameFormat=new TextFormat();
            gameFormat.color=0xcc0000;
            gameFormat.font="Verdana";
            gameFormat.size=10;
            gameInfo.defaultTextFormat=gameFormat;
        }
        public function setMove(pMove:String)
        {
            gameInfo.text=pMove;
        }
        public function getMove():String
        {
            return gameInfo.text;
        }
    }
}
```

Включенные сюда методы по получению и установке значительно упрощают доступ и отображение возможных для игрока ходов. Кроме того, текстовое поле может служить временным хранилищем значений, которые могут быть использованы в переменных или параметрах.

Контролеры кнопок

Класс кнопок создан с помощью классов `Sprite`, `TextField` и `TextFieldType`. В примере 13.7 показан сценарий этого класса.

Пример 13.7. MoveButton.as

```
package
{
    import flash.text.TextField;
    import flash.text.TextFieldType;
    import flash.display.Sprite;

    public class MoveButton extends Sprite
    {
        private var mvBtn:Sprite;
        private var id:String;

        public function MoveButton(mover:String,btncolor:int)
        {
            mvBtn=new Sprite();
            this.id=id;
            addChild(mvBtn);
            mvBtn.buttonMode=true;
            mvBtn.useHandCursor=true;
            mvBtn.graphics.beginFill(btncolor);
            mvBtn.graphics.drawRoundRect(0,0,(mover.length*8),20,5,5);
            var mvLbl:TextField=new TextField();
            mvBtn.addChild(mvLbl);
            mvLbl.text=mover;
            mvLbl.selectable=false;
        }
    }
}
```

Подобная комбинация классов `Sprite` и текстовых полей обеспечивает пользователю несколько больше информации, чем использование обычного класса кнопок `SimpleButton`. Поскольку мы решили в качестве информации о ходе пересылать текстовые строки и всю интерактивную информацию отображать в динамических текстовых полях, не меняя при этом внешний вид кнопок, создать данный класс было не очень сложно. Еще мы захотели немного поэкспериментировать с цветом кнопок, чтобы дизайнер интерфейса мог с их помощью реализовать свою собственную цветовую схему, поэтому мы включили в конструктор данного класса параметр цвета (`btncolor`).

Файл документа Flash и клипы связи

Последние детали оформления данного приложения включают в себя статический текст, размещенный прямо на его сцене, и мувиклипы, служащие для индикации соединения с сервером. Рисунок 13.5 показывает размещение статического текста в правой части сцены, а также общий вид нашего приложения в работе слева.

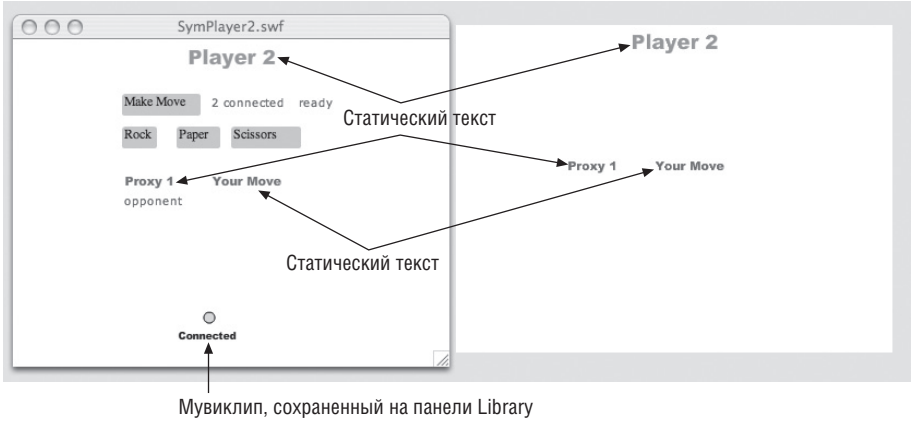


Рис. 13.5. *Вид сцены и swf-файла*

Следующие шаги помогут вам создать последнюю часть данного приложения:

1. Откройте новый файл Flash (ActionScript 3.0). Добавьте в него новый уровень. Назовите верхний уровень `Connect`, а нижний – `Text`.
2. В поле класс документа введите `SymPlayer1` и сохраните файл как *SymPlayer1.fla*.
3. Кликните на уровне `Text`, чтобы его выбрать. Используя инструмент для создания текста, выберите в нем статический текст, установите шрифт в Arial Black размером 18 точек цвета #666666. Введите надпись `Player1` и поместите ее по координатам X=158, Y=1.
4. Измените размер шрифта на 11 точек и напечатайте `Proxy 2` в позиции X=100, Y=120, а потом `You Move` в позиции X=180, Y=120. Заблокируйте уровень `Text`.
5. Кликните на уровне `Connect`, чтобы его выбрать. Используя инструмент для рисования овалов, выберите красный цвет для заполнения и черный цвет для линии, установите толщину линии в .25 и нарисуйте круг диаметром 10 точек.
6. Выберете нарисованный круг и нажмите `F8`, чтобы открыть окно конвертации в символ. В поле имени введите `Connect` и выберите в качестве типа символа `Movie clip`. Если вы увидите кнопку `Advanced`, то нажмите на нее, чтобы открыть панели `Linkage` и `Source`. Отметьте позицию экспорта в ActionScript на панели `Linkage`. Теперь вы должны увидеть текст «Connect» в поле класса и «flash.display.MovieClip» в поле базового класса. Щелкните на `OK`.
7. Кликните на `Frame 1` и откройте панель `Actions` (нажмите `F9` в Windows или `Option + F9` на Макинтоше). Введите `stop()` в панели `Actions`. Закройте панель.

8. Кликните на Frame 2 и нажмите F6, чтобы добавить второй ключевой фрейм. Измените цвет заполнения с красного на зеленый. Кликните на пиктограмме Scene 1, чтобы выйти из режима редактирования символа.
9. Вы должны увидеть мувиклип Connect на сцене. Удалите его со сцены. Откройте панель Library (выберите из меню Window → Library либо нажмите Ctrl + L в Windows или Command + L на Маке). Вы должны увидеть мувиклип Connect на панели Library. Убедитесь, что его имя начинается с большой буквы, поскольку это будет имя класса, которое вы будете использовать в классах SymPlayer1 и SymPlayer1.
10. Выберете из меню File → Publish, чтобы сгенерировать *swf*- и *html*-файлы.

В данном приложении предполагается наличие двух различных игроков, использующих два различных файла Flash. Чтобы создать второй файл, выполните следующее:

1. Откройте файл *SymPlayer1.fla* и, используя пункты меню File → SaveAs, сохраните его как файл *SymPlayer2.fla*.
2. Измените его класс документа на SymPlayer2.
3. Измените статические тексты с «Player 1» на «Player 2» и с «Proxy 2» на «Proxy 1». Сохраните файл.
4. Выберите из меню File → Publish, чтобы сгенерировать *swf*- и *html*-файлы.

Вот и все. Разместите *html*- и *swf*-файлы на сервере. Один игрок получает имя Player 1, а второй – Player 2; можно начинать игру. Вероятно, вы захотите создать небольшой *html*-файл, чтобы пользователь мог выбирать себе имя между Player 1 и Player 2, и потом соединять его с соответствующим игроком через *html*-файл. Если одно из имен будет уже занято, что покажет красный символ связи, то пользователь сможет просто уйти назад и переключиться на другое имя.

Заключение

Шаблон проектирования Симметричный заместитель представляет собой некоторый эксперимент в области шаблонов, начатый Эриком Гаммой и его коллегами, когда они писали книгу «Design Patterns: Elements of Reusable Object-Oriented Software». Основная идея Джеймса Хелиотиса и Акселя Шрейнера заключается в том, что игра для двух игроков, соединенных через Интернет, может быть сыграна прямо у клиентов с помощью симметричных судей и сведения как соперников локального игрока и заместителя оппонента на каждой из сторон. Это решение применимо не только для одной игры, но и для целого класса игр через Интернет.

Наша конкретная имплементация данного шаблона проектирования использовала Flash Media Server 2 для пересылки ходов каждой из сторон, и очень сомнительно, что Хелиотис и Шрейнер планировали такое использование FMS2 в своем шаблоне. В этом и проявилась сила данного шаблона; ведь шаблоны проектирования должны представлять собой общее решение некоторой проблемы, а не опираться на конкретные имплементации такого решения. Благодаря такой коммуникационной гибкости шаблона Симметричный заместитель мы смогли использовать при его реализации программу для обмена информацией через Интернет по своему выбору, а не указанную его разработчиками.

Встроив методы заместителя в общий интерфейс и имплементировав его прямо в двух игроках, мы смогли изменить общую схему шаблона проектирования, и это не нарушило его гибкости. Такое «усеченное» решение, возможно, несколько снижает гранулярность исходной схемы шаблона и увеличивает размеры его главных классов, но по-прежнему сохраняет структуру и общую идею данного шаблона. Шаблоны проектирования являются решениями общего вида, и пока они удовлетворяют критериям гибкости и повторного использования, одновременно реализуя лучшие практики применения ООП, они могут быть имплементированы бесконечным числом способов.

Алфавитный указатель

A

addChild(), метод, 232
Array, класс, 484
ASCII, 507
Atom Syndication Format, формат новостной ленты, 245

B

Bitmap, класс, 282
BtnState, класс, 432, 454
Button,класс, 30
Button, компонента, 310

C

change, событие, 503, 514
children, свойство, 252
colorFont, свойство, 152
Component, класс, 252, 259, 260
    пример, 254
component, ключевое слово, 177
Composite, класс, 252, 259, 260
    пример, 254, 256

D

DisplayObjectContainer, класс, 282, 283, 285
DisplayObject, класс, 282, 285, 286

E

E4X, 246
ECMAScript для XML, 246, 325
ECOOP, конференция, 540
enter frame, событие, 130, 232
Event.COMPLETE, событие, 269, 325
EventDispatcher, класс, 498
Event.SOUND_COMPLETE, событие, 269

F

File, меню, 310, 315
final, атрибут, 391, 397
final, ключевое слово, 552
Flash CS3, 14
Flash Media Server 2, 14, 16, 444, 446, 492, 555, 570
flash.media, пакет, 153
flash.net.URLRequest, класс, 153
Flex 2, 14, 16
for each, выражение, 248

G

getChild(), метод, 258
getComposite(), метод, 260
GoF (банда четырех), 15
graphics, свойство, 119, 127, 232

I

IllegalOperationError, ошибка, 255
InteractiveObject, класс, 284, 286
interface, ключевое слово, 340
internal, атрибут, 102, 105, 259
iPod фирмы Apple, 152

K

KeyboardEvent, событие, 501
key_down, событие, 236, 239, 276, 502

L

Leaf, класс, 252
    пример, 254, 255
Loader, класс, 180

M

mouse_down, событие, 130
mouse_move, событие, 130, 239
MovieClip,класс, 30, 117, 150, 151, 159, 204
mp3-файлы, воспроизведение, 153

N

NetBtn, класс, 432, 454
NetStream, класс, 429, 439, 447
new, ключевое слово, 95
Number, класс, 172

O

ObjectEncoding, класс, 445, 447
OOPSLA, конференция, 540
Open, меню, 310
operation(), метод, 265
override, ключевое слово, 56, 173, 175, 177, 398

P

parentNode, свойство, 259
Point, класс, 232, 276, 277
protected, атрибут, 103, 105, 259
public, атрибут, 259

R

Real-Time Messaging Protocol (RTMP), 459
RegExp, класс, 484
RSS, формат новостной ленты, 245, 323

S

SaveAs, меню, 310
Save, меню, 310, 315
Shape, класс, 282, 284, 286
SimpleButton, класс, 311
SoundChannel, класс, 153, 325
Sound, класс, 153
Sprite, класс, 117, 122, 127, 151, 230, 232, 271, 285, 286, 316, 325, 505
Sprite, объект класса, 30
Stage, класс, 122
Stage, объект, 232, 282, 316
static, атрибут, 267, 325
stopPropagation(), функция, 291
super, свойство, 127
super(), метод, 243, 286

T

TextField, класс, 150, 240, 241, 243, 313, 323
this, ключевое слово, 177
TimeEvent, сообщение, 519
Timeline, 153
trace(), функция, 347, 349, 401, 429, 436, 503

U

UIList, компонента, 377
Unicode, 176
URLLoader(), метод, 248

X

XML, язык, 245, 323, 325
xmlLoaded(), метод, 248

А

абстрактная функция, 391, 393, 394, 397, 400, 411, 413, 414, 416
абстрактность, 34
  концепция ООП, 151
  преимущества, 37
абстрактные классы, 56, 99, 169, 318, 391, 551
абстрактные методы, 98, 100, 255, 318, 471, 551
абстрактный автомат, 70
абстрактный декоратор, 173, 175
абстрактный интерфейс, 98
абстрактный класс, 52, 99, 100, 254
абстрактный компонент, 173
  описание, 174
абстракции, 35, 488
автоматный движок, 418, 424
алгоритм, 389, 390, 462, 464, 475, 479, 482, 488

Б

«банда четырех», 15, 19, 391, 393, 395, 410, 420, 463, 464, 492, 570
Буч, Гради, 34

В

вертикальная стрелковая игра, 122
взаимосвязь классов, 298
Влиссидес, Джон, 15, 19, 339
внутренние классы, 105

Г

Гамма, Эрик, 12, 15, 19, 339
гибкость, концепция ООП, 546
гибкость приложений, 339
Гири, Дэйвид, 463
глобальные переменные, 136
Голливуда принцип, 391, 393, 395, 402
гранулярность, 87, 478
грязные трюки в коде, 134, 189

Д

деградация зависимостей, 395
декоратор, абстрактный класс, 170, 173

декоратор, объект, 168
декорации, 176
декремент, 316
делегат, 465, 466, 480, 488
делегирование, 79, 465, 466, 468, 469, 470, 476, 477, 480, 487
Джонсон, Ральф, 12, 15, 19, 339
диаграмма классов, 136, 251, 301, 421, 424
диаграмма состояний, 418, 420, 437, 446, 460
диспетчер, 300, 301, 302, 303, 306, 310
  динамический, 334
  клавиши быстрого вызова, 315
  кнопочный, 310, 312, 314, 316, 326, 329, 332
  меню, 310, 315
дочерний узел, 251

Ж

жесткая связь классов, 134, 298

З

зависимости, 72
  классов, 96, 339
закрыт для модификаций, принцип ООП, 97
замещение метода, 56
запрос, 70
защищенный метод, 103

И

иерархическое дерево, 251
иерархия состояний, 419, 437
избыточное наследование, 343
имплементация, 69, 249, 387
имя экземпляра, 30
инверсная кинематика, 271
инвертированная структура контроля, 390, 395, 416
инкапсулируйте изменяемое, принцип ООП, 463
инкапсуляция, 38, 48, 93, 298, 306, 318, 334, 343, 464, 467, 488
инкремент, 316
интерфейс, 42, 43, 52, 97, 169, 220, 223, 249, 265, 340, 377, 387, 438
  команд, 299, 302, 318
    с отменой, 318
  узлов, 254
  макрокоманды, 307
  пользователя, 408
  продуктов, 101
использование вместо имплементаций, принцип ООП, 339, 468, 474, 555

К

«камень-ножницы-бумага», игра, 540
Кастило, Дейвид, 420
Кей, Джонатан, 420
кинематическая пара, 271, 276
кинематическая цепь, 271
  описание работы, 276
клавиши быстрого вызова, 315, 503
классы, 168
  Sprite, 282
  адаптера, 221, 222, 234
  агрегатор, 463, 465, 466
  декоратор, 172
  документа, 29, 100, 122
класс-создатель, 96
клиент, 70, 96, 103, 302, 304, 314, 480
код символа, 498, 507
командный объект, 300, 301, 304, 306, 310, 322
  динамический, 329, 331
  отмены, 320
  портативность, 315, 329, 334
композиция, 79, 222, 249, 338, 339, 342, 347, 391, 393, 479
  классов, 76
композиция вместо наследования, принцип ООП, 222, 229, 338, 339, 342, 393, 488
компонент для декорирования, 168, 170
  описание, 170
компоненты, 38, 250, 253
  Flash, 168
конверсия интерфейса, 233
конечный автомат, 418, 424, 427, 428, 429, 436, 444, 460
конкретные декораторы, 170, 172 – 174, 176
  описание, 170
конкретные команды, 299, 302, 306, 314, 327
  с отменой, 319
конкретные компоненты, 170, 172, 173, 174, 175
конкретные макрокоманды, 308
конкретные подкоманды, 307
конкретные функции, 411

конструктор, 257, 274, 280, 286, 328, 342, 347, 349, 351, 359
конструктор класса, 138, 140, 501
контейнер
  команды, 310
  отображаемых объектов, 282
координатная сетка, 381
«Космические захватчики», игра, 122

Л

линейный график, 374, 381
листовой узел, 251, 253

М

Майерс, Том, 26
макрокоманда, 306, 334
  интерфейс, 307
  подкоманда, 307
марионетка, 271
медиа-файл, 323
метаморфозы, 60
метод получатель, 43, 426
метод установщик, 358, 426, 481
многократное использование кода, 58
множественное наследование, 225, 226
множественные шаблоны, 491
мобиль, игрушка детская, 252
модули, 38

Н

наследование, 48, 168, 171, 222, 225, 229, 249, 339, 342, 390, 391, 393, 466, 488
  имплементации, 226
  интерфейса, 226
  нежелательное, 171
  приватное, 226
  публичное, 226
Нахимовский, Александр, 26
новостная лента, 245, 323

О

область видимости, 27
объект, 168
объект макрокоманды, 309
один-ко-многим, отношение, 337
оконечная компонента, 504
оконечный узел, 251
ООП (объектно-ориентированное программирование), 25, 33
  базовые концепции, 33
  принципы, 97
опорная точка Sprite, 232
открыт для расширений, принцип ООП, 97
отмена команд, 317
отношение, 79
отображаемый объект, 282
отображение информации, 377

П

пакет, 100, 138
параллельные иерархии классов, 115
параметры
  методов, 96
  фабричного метода, 111
параметризованный конструктор, 229, 231, 236, 301, 302
параметризованный фабричный метод, 111
первородный грех, 190
переход между состояниями, 419, 422
поддержка приложений, 85, 86
подкаст, 323
подкласс, 50, 171, 172, 390
подкоманда, 306
полиморфизм, 60, 265, 421, 422, 488
портативность, 315
последовательное программирование, 26
преимущество композиции, 76
приватные функции, 137
приватный класс, 138, 139
приватный конструктор, 137, 138
примеры, 223, 226
  макрокоманда, 306
  манипулятор числами, 310
  минимальный, 302
  отмена команд, 317
  подкаст радио, 322
  поручения мамы, 298
  тематическое подкаст радио, 328
примитивный объект, 265
примитивный узел, 253
принцип Голливуда, 391, 393, 395, 402
проверка
  коллизий, 123
  попаданий, 123
продукт, класс, 96
прозрачный ящик, 343
простая компонента, 271
простая фабрика, 97
процедурное программирование, 27
публикатор новостной ленты, 245

публичные классы, 102
публичный метод, 105

Р

разделение команд, 317, 334
распространение событий, 291
расширяемость приложений, 87
расширяемые, но не изменяемые классы, 173
регулярные выражения, 484
родительский узел, 251

С

сборщик мусора, 258, 260, 262
  Flash, 124
свободный конец кинематической цепи, 271
связь классов, 96
сервер, 70
сигнатура
  метода, 228, 265
  объекта, 43
симуляторы устройств, 418
слабая связь классов, 306, 337
слабосвязанное проектирование, 116
слабосвязанность, принцип ООП, 510
создатель, класс, 96, 102
сортировка, 479, 484
составная компонента, 506
составной класс, 504
составной объект, 251, 265
составной узел, 251, 253
состояние, 69, 337
спагетти-код, 27
список отображения, 121, 282, 293, 314, 332
  мелодий, 265
  пример с самолетом, 282
стек команд, 318
  структура данных, 318
столбчатая диаграмма, 374, 379
структурные шаблоны проектирования, 165
сценарии, 28
суперкласс, 50, 171, 286
супертип, 340, 387, 391

Т

Тайдвел, Дженифер, 88
таймер, 519, 535
Тафт, Эдвард, 381
текстовые поля динамические, 485
трансформация типа данных, 172
триггер, 407, 420
  команды, 304, 312

У

увеличение числа в текстовом поле, 313
удаление узлов, 259, 261
уменьшение числа в текстовом поле, 313
унаследованный объект, 221
упаковка, 171
управление зависимостями, 117, 134

Ф

фабрика, метод, 96
фабрика спрайтов, 117
фабричный метод, 102
фиксированная функция, 399, 402
фиксированный метод, 391
Фриман, Элизабет, 395
Фриман, Эрик, 395, 463
функция-получатель, 177
функция-установщик, 350

Х

Хелиотис Джеймс, 540
Хелм, Ричард, 12, 15, 19, 339

Ц

целевой интерфейс, 238, 249

Ч

черный ящик, 38

Ш

шаблоны проектирования, 23, 25, 68
  Адаптер, 221
    адаптер класса, 222, 225, 229
    адаптер объекта, 222, 223, 229
    основные характеристики, 222
    отображение списка, 239
    существующий класс, 222, 249
    существующий объект, 221
  Враппер, 171
  Декоратор, 166, 492
    базовый класс, 173, 174
    диаграмма классов, 170
    дополнительные возможности, 173, 175, 218
    ключевые концепции ООП, 171

минимальный пример, 173
недостатки, 219
область применения, 218
описание, 167, 218
основная задача, 171
основные характеристики, 168
пример, 180, 188, 206
состав классов, 173
упаковывание, 168, 169, 171, 172, 178
Компоновщик, 250
доступ к узлам, 258
имплементация по умолчанию, 254
инверсная кинематика (змейка), 270, 281 методы, 252
свойства, 251, 293
структура, 251, 253, 257
Наблюдатель, 395
диаграмма классов, 338
идентификация наблюдателей, 349, 351
интерфейс
наблюдателя, 344, 345
рассылки, 344
ключевые концепции ООП, 339
множественные состояния, 350
описание, 335, 337, 387
основная задача, 387
основные характеристики, 337
отказ от подписки, 356
отписка, 338, 358
подписка, 338, 358
пример, 344, 349, 358, 374
рассылка, 337, 338, 344, 345, 352, 356, 358, 367, 374, 375
состояние, 338, 349, 356, 358
свойства, 337, 338, 344, 347, 351, 356, 358, 359, 367, 375
Состояние, 417, 462, 463, 488
интерфейс состояния, 417, 421, 426, 427, 437, 439, 460
класс состояния, 421, 422, 426, 439, 445, 450, 451, 460
ключевые концепции ООП, 421
контекстный класс, 422, 424, 426, 431, 440, 446, 452, 460
модель, 418
описание, 417, 422, 460
особенности, 418
переход, 417, 419, 422, 428, 437, 448, 449, 450, 460
пример, 422, 429, 436, 444
состояния, 417
увеличение числа, 424, 441
структура классов, 420, 422
Стратегия, 420, 422
диаграмма классов, 462, 489
добавление стратегий, 475, 478, 487
интерфейс, 465, 467, 471, 473, 478, 480, 481, 482
класс контекста, 465, 466, 468, 470, 475, 477, 478, 480, 481
класс стратегии, 465, 466, 467, 470, 471, 475, 483
ключевые концепции ООП, 463
ключевые характеристики, 462
описание, 461, 464, 465, 488
организация стратегий, 478
пример, 466, 470, 479
Шаблонный метод, 388
абстрактный класс, 396
диаграмма классов, 392
ключевые концепции ООП, 393
ключевые характеристики, 390
конкретные подклассы, 401, 404
конкретный подкласс, 396
операции, 391, 396, 397, 403, 405, 411
описание, 388, 395, 416
основной алгоритм, 391, 392, 395, 397, 399, 403, 405, 411, 413, 416
подключаемые операции, 391, 393, 410, 412, 414, 416
пример, 396, 399, 403, 410
Фабричный метод, 95, 395
рекомендации по использованию, 148
ситуации применения, 122
Модель-Представление-Контроллер, 491
вложенные представления, 507
карты погоды, 511
инфракрасные, 519
ключевые особенности, 510
контроллер, 494, 500, 511, 514, 522, 528, 535, 537
машинки, 526
машинка-преследователь, 535
минимальный, 498
модель взаимодействия, 495, 499, 511, 521, 526, 537
описание, 493, 511, 526, 537

представление, 494, 501, 511, 515, 523, 530, 532, 534, 537
рассылка, 498
стратегия, 492, 498, 500, 501, 510, 537, 538
Симметричный заместитель, 540, 570
диаграмма объектов, 543
заместитель игрока, 543, 558
игровая гибкость, 547
игрок, 541
интерфейс, 541, 549
коммуникационная гибкость, 547, 571
разделяемые объекты FMS2, 555
судья, 541, 550, 565
ячейки синхронизации, 545
Шрейнер Аксель, 540

Э

экономия пакетов, концепция ООП, 151

Я

«является», отношение классов, 229

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-189-9, название «ActionScript 3.0. Шаблоны проектирования» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, атакже сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.