МАРК ГРАНД

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ В JAVA

- **Каталог** *популярных шаблонов проектирования, проиллюстрированных при помощи UML*

Patterns in Java™, Volume 1

A Catalog of Reusable Design Patterns Illustrated with UML

Second Edition

MARK GRAND

Wiley Publishing, Inc.

МАРК ГРАНД

ШАБЛОНЫ проектирования в JAVA

Каталог
популярных шаблонов проектирования, проиллюстрированных при помощи UML

МОСКВА ООО «НОВОЕ ЗНАНИЕ» 2004

УДК 004.451
ББК 32.973.26-018.2
Г77

Перевод с английского *С. Беликовой*

Гранд М.

Г77 Шаблоны проектирования в Java / М. Гранд; Пер. с англ. С. Беликовой. — М.: Новое знание, 2004. — 559 с.: ил.

ISBN 5-94735-047-5 (рус.).
ISBN 0-471-22729-3 (англ.).

Подробно рассмотрено применение в Java шаблонов проектирования (patterns), которые представляют собой многократно используемые решения широко распространенных проблем. Продемонстрировано, каким образом применение шаблонов повышает производительность работы программистов — и профессионалов, и начинающих. Приведен обзор языка UML и описано 47 наиболее важных шаблонов проектирования.

Книга предназначена для программистов, разрабатывающих приложения на Java.

УДК 004.451
ББК 32.973.26-018.2

ISBN 5-94735-047-5 (рус.)
ISBN 0-471-22729-3 (англ.)

ОГЛАВЛЕНИЕ

Об авторе

Марк Гранд, живущий в Атланте, — консультант с более чем 20-летним стажем. Он специализируется в области распределенных систем, объектно-ориентированного проектирования и языка программирования Java. Он создавал архитектуру первого коммерческого программного продукта для электронной коммерции в интернете. В настоящее время он работает над открытой оболочкой, реализующей идею интеграции компонент и программ в пользовательское приложение.

Самую большую известность Марку Гранду принес его бестселлер «Шаблоны проектирования в Java». Помимо преподавания языка Java в компании Sun и образовательных учреждениях, Марк принимал участие в некоторых крупномасштабных коммерческих проектах, реализованных на Java.

До того как начать заниматься Java, Марк в течение 11 лет работал проектировщиком и разработчиком языков программирования четвертого поколения (4GL). Он занимал должность архитектора и руководителя проекта по выпуску продукции взаимного обмена электронными данными. Марк работал в организациях, занимающихся разработками в области информационных технологий, выполняя функции архитектора программного обеспечения, баз данных, проектировщика и администратора сетей, а также системного администратора в компании Sun. Начиная с 1982 года, он занимается объектно-ориентированным программированием и проектированием.

ГЛАВА 1

Введение в шаблоны проектирования

Шаблоны проектирования — это многократно используемые решения широко распространенных проблем, возникающих при разработке программного обеспечения (ПО). Поскольку вся книга посвящена шаблонам проектирования, далее они называются просто *шаблоны*.

По мере приобретения опыта программисты признают сходство новых проблем с решаемыми ими ранее. С накоплением еще большего опыта приходит осознание того, что решения похожих проблем представляют собой повторяющиеся шаблоны. Зная эти шаблоны, опытные программисты распознают ситуацию их применения и сразу используют готовое решение, не тратя время на предварительный анализ проблемы.

Когда программист создает нужный шаблон, это заслуга его интуиции. В большинстве случаев путь от несформулированного интуитивного представления до хорошо продуманной идеи, которую программист может четко выразить словами, на удивление труден. Это очень важный этап, потому что если программист настолько хорошо понимает шаблон проектирования, что может описать его словами, то он в состоянии разумно комбинировать его с другими шаблонами. Еще важнее то, что однажды сформулированный шаблон может использоваться и другими программистами, знакомыми с этим шаблоном. Это позволяет программистам эффективнее сотрудничать и объединять свои интеллектуальные возможности. Кроме того, это помогает избежать ситуации, когда программисты, обсуждая различные решения проблемы, только потом обнаруживают, что они имели в виду одно и то же решение, но выраженное разными способами. Также, если шаблон описан, то более опытные программисты могут научить пользоваться им тех программистов, которые с ним не знакомы.

Основная цель этой книги — предоставить программистам описание наиболее используемых шаблонов проектирования. Кроме того, благодаря этой книге программисты могут сами открывать новые шаблоны.

Хотя в книге представлено довольно много шаблонов проектирования, существуют и другие шаблоны. Некоторые из них могут показаться нужными только небольшому кругу людей. Другие могут вызывать большой интерес. Читатели, желающие обсудить такие шаблоны, могут связаться с автором по электронной почте: mgrand@mindspring.com.

Рассмотренные в книге шаблоны представляют собой конструктивные способы организации некоторых частей цикла разработки ПО. Существуют другие шаблоны, которые периодически повторяются в программах, но не являются конструктивными. Они называются *антишаблонами*, и так как могут свести на нет всю пользу от применения шаблонов, то в этой книге они не описываются.

Шаблоны и антишаблоны кажутся похожими, но на самом деле различны по своей сути. Цель шаблона — распознать возможность применения хорошего решения проблемы. Назначение же антишаблона в том, чтобы выяснить суть плохой ситуации и предложить решения.

Описание шаблонов

Обычно шаблон описывается по следующей схеме:

- имя шаблона, под которым он широко известен. Если у шаблона несколько имен, то они тоже приводятся;
- описание проблемы, которое включает конкретный пример и решение, специально предназначенное для данной проблемы;
- краткое изложение рассуждений, приводящих или к формулированию общего решения, или к выводу о его неприменимости;
- общее решение;
- последствия, хорошие и плохие, как результат использования данного решения проблемы;
- перечень шаблонов проектирования, связанных с данным шаблоном.

Книги, посвященные шаблонам, различаются по способу представления этой информации. Все представленные в данной книге шаблоны относятся к стадии проектирования жизненного цикла программы. Описание шаблонов организовано в виде следующих разделов.

ИМЯ ШАБЛОНА

Заголовок параграфа содержит имя шаблона. У большинства шаблонов после названия дополнительный текст отсутствует. Если текст есть, то он представляет собой другие имена шаблона или информацию о его происхождении и общих характеристиках.

СИНОПСИС

В этом разделе приводится краткое описание шаблона. Синопсис передает суть решения, предоставляемого шаблоном. Синопсис предназначен, в первую очередь, для опытных программистов, которые могут знать сам шаблон, но не знать его название

Вас не должно обескураживать то, что вы не распознали шаблон по имени и синопсису. Внимательно прочтите остальную часть описания шаблона и попытайтесь понять его.

КОНТЕКСТ

Здесь описывается проблема, для решения которой предназначен шаблон. Для большинства шаблонов проблема представлена в виде конкретного примера. После описания проблемы предлагается ее решение.

МОТИВЫ

Здесь приводятся соображения с целью выработки основного решения, представленного в разделе «Решение». Кроме того, здесь могут быть указаны причины, по которым решение не используется:

☺ — таким символом обозначены причины, лежащие в основе использования решения;

☹ — этим значком помечаются причины отказа от применения решения.

РЕШЕНИЕ

Данный раздел — основная часть шаблона. Здесь описывается универсальное решение проблемы, которое предоставляет шаблон.

РЕАЛИЗАЦИЯ

В данном разделе описываются важные соображения, которые нужно учитывать при использовании решения. Здесь могут быть описаны также некоторые широко распространенные или упрощенные варианты решения.

СЛЕДСТВИЯ

Здесь объясняются последствия, хорошие и плохие, являющиеся результатом использования данного решения. Большая часть следствий помечена маркерами следующего вида:

☺ — положительные следствия;

• — нейтральные;

☹ — отрицательные.

ПРИМЕНЕНИЕ В JAVA API

Если в ядре Java API есть соответствующий пример использования шаблона, т

ПРИМЕР КОДА

Данный раздел содержит пример исходного кода, демонстрирующего реализацию проекта, который использует шаблон. В большинстве случаев это проект, описанный в разделе «Контекст».

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ДАННЫМ ШАБЛОНОМ

В этом разделе приводится список шаблонов проектирования, связанных с описываемым шаблоном.

Краткая история шаблонов проектирования

Идея шаблонов проектирования первоначально возникла в архитектуре. Архитектор Кристофер Александер написал две революционные книги, содержащие описание шаблонов в строительной архитектуре и городском планировании: «A Pattern Language: Towns, Buildings, Constructions» (Oxford University Press, 1977) и «The Timeless Way of Building» (Oxford University Press, 1979). Идеи, представленные в этих книгах, могут применяться и в других областях, не имеющих отношения к архитектуре, в том числе и в области разработки ПО.

В 1987 году Уард Каннингэм и Кент Бек использовали некоторые идеи Александера для разработки пяти шаблонов при проектировании интерфейса пользователя (user interface, UI). Они опубликовали статью «Using Pattern Languages for Object-Oriented Programs», посвященную шаблонам при проектировании интерфейса пользователя, в OOPSLA-87.

В начале 1990-х годов Эрих Гамма, Ричард Хелм, Джон Влиссидес и Ральф Джонсон начали работу над книгой «Design Patterns» — одной из наиболее выдающихся книг того десятилетия в компьютерной области. Опубликованная в 1994 году книга популяризировала идею шаблонов. Книгу «Design Patterns» часто называют «Gang of Four» (GoF).

Эта книга представляет собой второе издание, в которое включены некоторые дополнительные шаблоны. Шаблоны были усовершенствованы на основе предложений читателей первого издания. Кроме того, были переработаны примеры с учетом изменений в языке Java.

Книга «Шаблоны проектирования в Java» отражает эволюцию шаблонов и объектов со времени опубликования книги GoF, в которой примеры написаны на языках C++ и Smalltalk. В данной книге используется язык Java, и почти все объекты рассматриваются с точки зрения Java. При написании книги «Gang of Four» UML (Unified Modeling Language, унифицированный язык моделирования) еще не существовал. Сейчас он получил широкое распространение как самый популярный язык для объектно-ориентированного анализа и проекти-

Структура книги

Эта книга посвящена шаблонам проектирования, которые используются н микроархитектурном уровне.

Она начинается с описания средств языка UML. Глава 3 содержит обзор жи ненного цикла ПО и знакомит читателей с контекстом использования шабл нов. Далее в этой главе описывается общий пример применения шаблоно В остальных главах рассматриваются различные типы шаблонов.

Сайт http://mgrand.home.mindspring.com содержит синопсис шаблонов и пр меры исходных кодов, включенных в данную книгу.

Представленные здесь примеры кода написаны на языке Java, версия 1.4.

ГЛАВА 2

Обзор UML

UML (Unified Modeling Language, унифицированный язык моделирования) — это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Данная глава содержит краткий обзор UML, в ходе которого можно ознакомиться как с подмножеством, так и с расширениями UML, используемыми в книге. Полное описание UML можно найти на сайте http://www.omg.org/technology/documents/formal/uml.html.

В книгах о UML некоторые данные, хранящиеся в экземплярах класса, называются *атрибутами*; а *операциями* называется инкапсулированное поведение классов. Такие термины не связаны с конкретным языком реализации. В этой книге предполагается, что в качестве языка реализации вы используете Java. Кроме того, в книге применяется в основном не нейтральная терминология, менее знакомая программирующим на Java, а связанная с Java. Например, слова «атрибут» и «переменная» — взаимозаменяемые, причем предпочтение отдается характерному для языка Java термину «переменная». Взаимозаменяемыми являются здесь также слова «операция» и «метод», но предпочтение отдается термину «метод».

UML определяет ряд диаграмм различного вида. Они описаны в этой главе. Если у вас есть опыт объектно-ориентированного проектирования, вам покажется знакомой большая часть понятий, лежащих в основе системы обозначений UML. Если в какой-либо другой главе некоторые понятия окажутся незнакомыми, то вам достаточно возвратиться к этой главе.

Диаграмма классов

Диаграмма классов — это диаграмма, на которой показаны классы, интерфейсы и отношения между ними. Главный элемент диаграммы классов — *класс*. На рис. 2.1 приведен пример класса и его характеристики, которые могут ото бражаться на диаграмме классов.

Классы изображаются в виде прямоугольников, обычно разделенных на две или три части. Прямоугольник класса на рис. 2.1 разделен на три части. В верхней части находится имя класса. Средняя часть содержит список переменных класса, а нижняя часть — методы класса.

Символы, указанные перед каждой переменной и каждым методом, представ ляют собой *индикаторы видимости* (visibility indicators). Возможные индикаторы видимости и их значения содержатся в табл. 2.1.

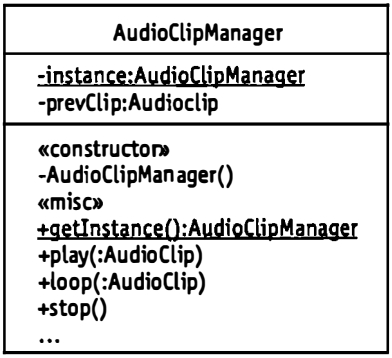


Рис. 2.1. Основная схема класса

Переменные, изображенные в средней части прямоугольника, имеют вид

```
visibilityIndicator name : type
```

Очевидно, что две переменные, показанные внутри класса, являются закрытыми (private). Имя первой переменной — `instance`, ее тип — `AudioClipManager`. Имя второй — `prevClip`, ее тип — `AudioClip`.

Хотя это и не показано на рис. 2.1, переменной может присваиваться соответствующее ее типу начальное значение при помощи знака «=» и некоторой величины:

```
shutDown:boolean = false
```

Таблица 2.1. Индикаторы видимости

Индикаторы видимости	Значение
+	открытый (Public)
#	защищенный (Protected)
-	закрытый (Private)

Заметим, что первая переменная, изображенная в классе, подчеркнута. Это означает, что она является статической. Данное утверждение справедливо и для методов.

Методы, показанные в нижней части прямоугольника, имеют вид

```
visibilityIndicator name ( formalParameters ) : returnType
```

Метод `getInstance`, представленный в классе на рис. 2.1, возвращает объект `AudioClipManager`.

Если метод ничего не возвращает, то в UML из *сигнатуры*[1] метода убирается `:returnType`, как у метода `stop` на рис. 2.1.

Формальные параметры (formal parameters) метода состоят из имени и типа:

```
setLength (length:int)
```

Если метод имеет несколько параметров, то они отделяются друг от друга запятой:

```
setPosition (x:int, y:int)
```

Упомянутый выше класс содержит два метода, перед которыми указано слово в угловых кавычках, например:

```
«constructor»
```

На схемах UML слово, заключенное в угловые кавычки, называется *стереотипом*. Стереотип описывает то, что за ним следует. Стереотип `constructor` указывает на то, что следующие за ним методы представляют собой конструкторы. Стереотип `misc` указывает, что следующие за ним методы — регулярные.

Последний элемент, изображенный в классе на рис. 2.1, представляет собой эллипсис (...). Если в нижней части прямоугольника класса указан эллипсис, то класс имеет дополнительные методы, которые на диаграмме не обозначены. Если эллипсис находится в средней части класса, то класс имеет дополнительные переменные, не показанные на диаграмме.

Как правило, нет необходимости (или просто неудобно) показывать столько деталей класса, как на рис. 2.1. Изображение класса может состоять только из двух частей (рис. 2.2). В таком случае верхняя часть содержит имя класса, а нижняя — методы. Переменные класса при этом просто не показаны. Это не означает, что класс их не имеет.

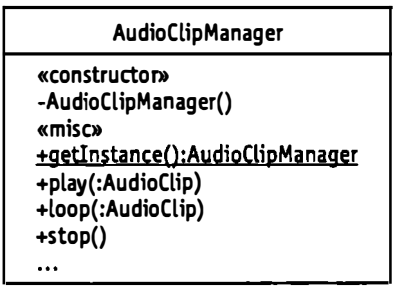


Рис. 2.2. Класс, состоящий из двух частей

Как у методов, так и у переменных могут отсутствовать индикаторы видимости. Если метод или переменная изображены без индикатора видимости, то это означает отсутствие указания видимости метода или переменной. При этом не

[1] Сигнатура метода — это комбинация его имени и формальных параметров.

подразумевается, что методы или переменные являются открытыми (public), защищенными (protected) или закрытыми (private).

Параметры метода могут быть опущены, если возвращаемые ими переменные не указаны. Например, у класса на рис. 2.3 отсутствуют индикаторы видимости и параметры методов.

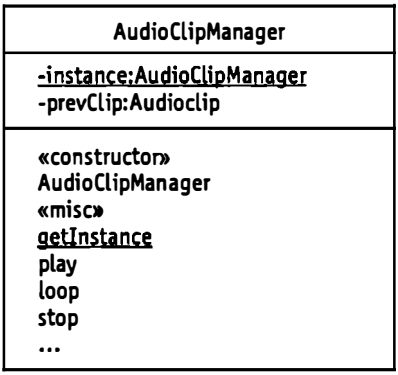


Рис. 2.3. Упрощенная схема класса

На рис. 2.4 представлена простейшая схема класса, состоящая только из одной части, содержашей имя класса. Такое упрошенное представление просто идентифицирует класс. В этом представлении отсутствует информация о переменных и методах класса.

Рис. 2.4. Схема класса, состоящая из одной части

Интерфейсы изображаются так же, как классы. Единственное отличие состоит в том, что перед указанным в верхней части именем указан стереотип `interface` (рис. 2.5).

На рис. 2.6 показан абстрактный класс `Product`, являюшийся суперклассом для класса `ConcreteProduct`. Написание имени класса курсивом говорит о том, что этот класс — абстрактный. Абстрактные методы класса также выделены курсивом.

Линии на рис. 2.6 указывают на взаимосвязь между классами и интерфейсом. Сплошная линия со стрелкой в виде замкнутого контура (как линия на рис. 2.7) указывает на то, что данный подкласс наследуется от суперкласса.

Сушествуют линии, указываюшие, что класс реализует интерфейс. В этом случае используется пунктирная или штрих-пунктирная линия со стрелкой в виде замкнутого контура (рис. 2.8).

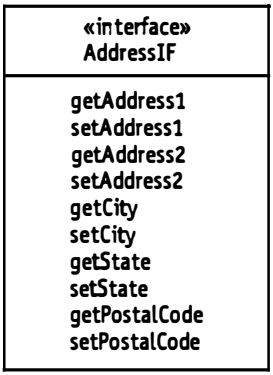


Рис. 2.5. Интерфейс

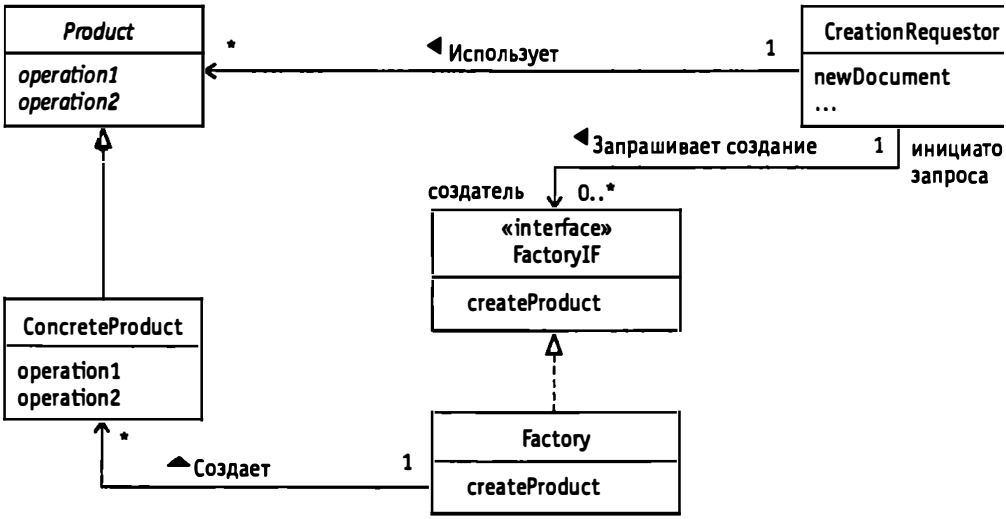


Рис. 2.6. Диаграмма классов

Рис. 2.7. Подкласс наследуется от суперкласса

Рис. 2.8. Класс реализует интерфейс

На рис. 2.6 изображен класс `Factory`, реализующий интерфейс `FactoryIF`.

Другие линии отображают иные типы отношений между классами и интерфейсом. В UML отношения такого типа называются *ассоциациями*. Вместе с ассоциациями могут указываться некоторые данные, представляющие информацию о сути ассоциации. Хотя следующие элементы не являются обязательными, но в данной книге они используются в тех случаях, когда это имеет смысл.

Имя ассоциации. Примерно в середине ассоциации может быть указано ее имя, которое всегда начинается с заглавной буквой. В конце или в начале имени ассоциации может быть изображен треугольник. Он указывает направление, в котором следует читать ассоциацию.
Например, на рис. 2.6 изображена ассоциация Создает между классами `Factory` и `ConcreteProduct`.

Стрелки навигации. Стрелки на концах ассоциаций называются *стрелками навигации*. Они указывают направление перемещения по ассоциации.

Взглянув на ассоциацию Создает (рис. 2.6), можно увидеть, что стрелка навигации направлена от класса `Factory` к классу `ConcreteProduct`. Принимая во внимание принципы создания объектов, кажется очевидным указание ответственности класса `Factory` за создание экземпляров объектов класса `ConcreteProduct`.

Природа некоторых других ассоциаций менее очевидна. Для понимания сути таких ассоциаций может возникнуть необходимость в предоставлении дополнительной информации, касающейся ассоциации. Обычно это делается так: задается имя роли, которую каждый класс играет в ассоциации.

Имя роли. Для описания смысла ассоциации имя роли, которую каждый класс играет в ассоциации, может быть указано на любом конце ассоциации, ближайшем к соответствующему классу. Имена ролей всегда записывают строчными буквами. Это позволяет легко отличать их от имен ассоциаций, всегда начинающихся с заглавной буквы.

Диаграмма классов, изображенная на рис. 2.6, содержит класс `CreationRequestor` и интерфейс `FactoryIF`, которые участвуют в ассоциации Запрашивает создание. Класс `CreationRequestor` участвует в ассоциации в роли инициатор запроса. Интерфейс `FactoryIF` участвует в ассоциации в роли создатель.

Индикатор множественности. Другая часто встречающаяся характеристика ассоциации сообщает о том, сколько экземпляров каждого класса участвуют в ассоциации. Индикатор множественности, содержащий такую информацию, может указываться на любом конце ассоциации. Он представляет собой просто число (например, 0 или 1) или диапазон чисел, который обозначается примерно так:

```
0..2
```

Звездочка, используемая вместо верхней границы диапазона, означает неограниченное количество случаев. Индикатор множественности 1..* указывает по крайней мере на один экземпляр, 0..* — на любое количество экземпляров.

Просто * эквивалентна 0..*. Взглянув на индикаторы множественности, представленные на рис. 2.6, можно увидеть, что все ассоциации схемы представляют собой соотношение «один ко многим».

На рис. 2.9 представлена диаграмма классов, на которой показан класс с множеством подклассов.

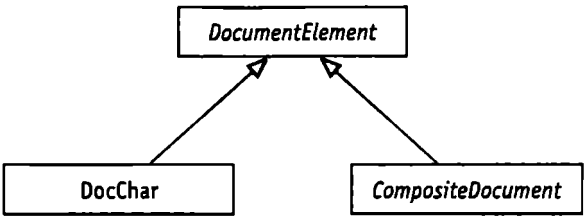


Рис. 2.9. Несколько стрелок наследования

Хотя рис. 2.9 справедлив, UML позволяет использовать более приятный с эстетической точки зрения способ изображения класса с множеством подклассов. Как показано на рис. 2.10, стрелки можно объединять. По смыслу рис. 2.10 ничем не отличается от диаграммы, представленной на рис. 2.9.

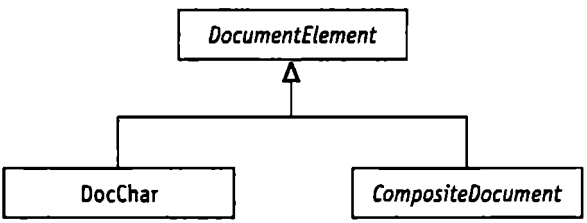


Рис. 2.10. Одна стрелка наследования

Иногда может возникнуть необходимость большей структуризации, чем на схемах, простым соотношением «один ко многим». Соотношение «один ко многим», в котором один объект содержит набор других объектов, называется *агрегацией*. На агрегацию указывает контур в виде ромба, расположенный на том конце ассоциации, который стыкуется с классом, содержащим экземпляры другого класса (рис. 2.11).

Рис. 2.11. Агрегация

На рис. 2.11 изображен класс `MessageManager`. Каждый его экземпляр содержит ноль или более экземпляров класса `MIMEMsg`.

В UML можно использовать еще одно обозначение, указывающее на более сильную, чем агрегация, связь. Эта связь называется *композитной агрегацией*. Чтобы агрегация была композитной, должны выполняться два условия:

- в какой-то момент времени агрегируемые экземпляры должны принадлежать только одному составному объекту;
- некоторые операции должны передаваться по наследству от составного объекта к его агрегируемым экземплярам. Например, если составной объект клонируется, то обычно клонируются и его агрегируемые экземпляры, поэтому получившийся составной объект обладает собственными клонами исходных агрегированных экземпляров.

На рис. 2.12 представлена диаграмма классов, содержащая композитную агрегацию. Объекты класса `Document` могут содержать объекты класса `Paragraph`, которые могут иметь в своем составе объекты класса `DocChar`. Благодаря композитной агрегации очевидно, что объекты класса `Paragraph` не используют совместно объекты класса `DocChar`, а объекты класса `Document` не используют совместно объекты класса `Paragraph`.

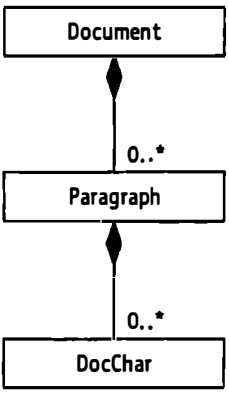


Рис. 2.12. Композитная агрегация

Некоторые ассоциации являются косвенными. Вместо прямого связывания классов друг с другом они связываются косвенно через третий класс. Рассмотрим диаграмму классов, представленную на рис. 2.13.

Ассоциация показывает, что экземпляры класса `Cache` ссылаются на экземпляры класса `Object` через экземпляр класса `ObjectID`.

На диаграмме классов эллипсис может иметь значение, отличное от показанного на рис. 2.2. На некоторых диаграммах нужно указать, что класс имеет большое или неопределенное множество подклассов, показывая при этом только несколько подклассов в качестве характерного примера (рис. 2.14).

Класс `DataQuery` имеет подклассы `JDBCQuery`, `OracleQuery`, `SybaseQuery` и неопределенное количество других классов, которые обозначены эллипсисом.

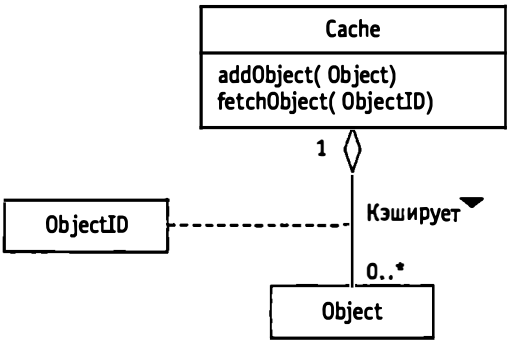


Рис. 2.13. Класс ассоциации

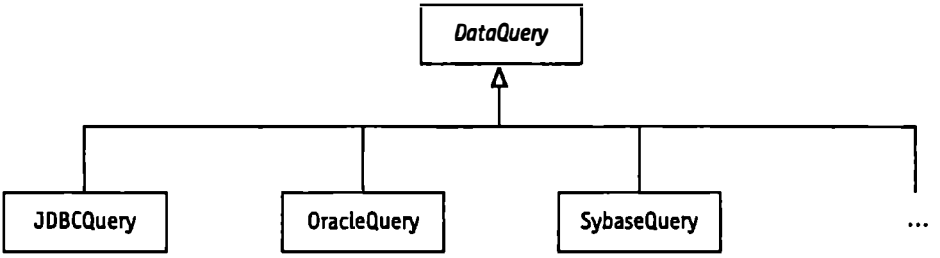


Рис. 2.14. Неопределенное количество подклассов

Ассоциация между классами или интерфейсами подразумевает наличие зависимости, которая содержит объектную ссылку, связывающую два объекта Имеется в виду, что один класс или интерфейс содержит в себе ссылку на другой класс или интерфейс, и это взаимодействие отражается на диаграмме. Возможны также зависимости других видов. Для указания зависимости более общего вида используется пунктирная линия. Пример подобной зависимости представлен на рис. 2.15.

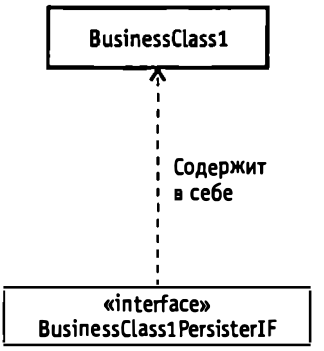


Рис. 2.15. Зависимость

Классы на диаграмме могут быть организованы в *пакеты*. Пакет изображается как большой прямоугольник с маленьким прямоугольником наверху, в котором указывается имя пакета (рис. 2.16).

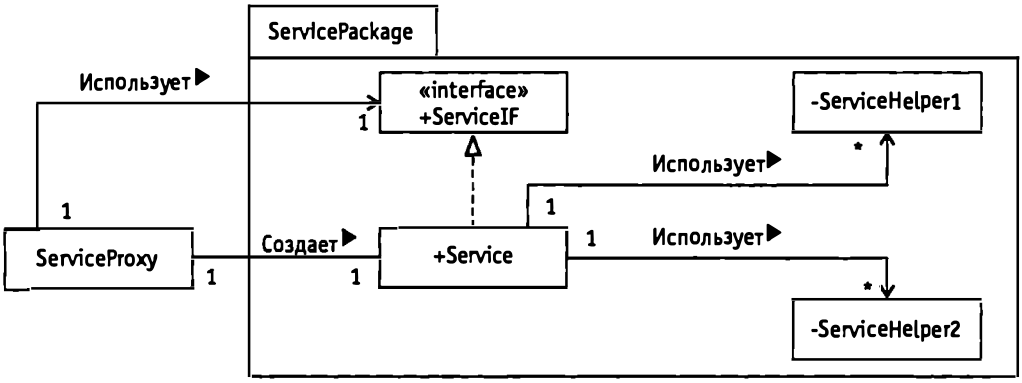
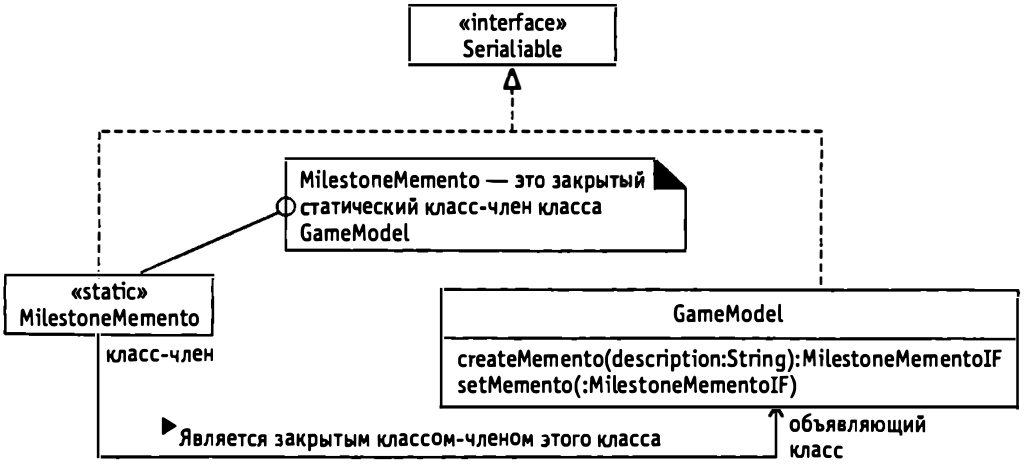


Рис. 2.16. Пакет

На рис. 2.16 показан пакет с именем `ServicePackage`. Перед именем класса или интерфейса, который находится внутри пакета, может указываться индикатор видимости. Открытые классы доступны для классов за пределами пакета, закрытые — нет.

Иногда при проектировании обнаруживаются обстоятельства, непонятные без комментария на диаграмме. В UML комментарий изображается в виде прямоугольника с загнутым правым верхним углом. С помощью сплошной линии комментарии присоединяются к тому элементу диаграммы, к которому они относятся (рис. 2.17).



В UML нет стандартного способа представления статического закрытого класса-члена. Чтобы указать статичность класса `MilestoneMemento`, на диаграмме используется стереотип в качестве расширения UML. Ассоциация указывает на то, что класс `MilestoneMemento` является закрытым членом класса `GameModel`. Чтобы сделать отношение еще более явным, на диаграмме классов, представленной на рис. 2.17, присутствует соответствующий комментарий.

Диаграммы классов могут содержать объекты. В данной книге большая часть объектов, указанных на диаграммах классов, представлена так, как на рис. 2.18. Изображенный здесь объект — это экземпляр класса `Area`. Подчеркивание свидетельствует о том, что это — объект. Слева от имени может находиться двоеточие «:». Единственное назначение имени состоит в том, что его можно использовать для идентификации определенного объекта.

Рис. 2.18. Объект

На некоторых диаграммах объект изображается в виде пустого прямоугольника. Очевидно, что пустые объекты не могут использоваться для идентификации объекта какого-либо определенного типа. Но они могут быть использованы на диаграмме, которая показывает структуру связей объектов неопределенного типа (рис. 2.19).

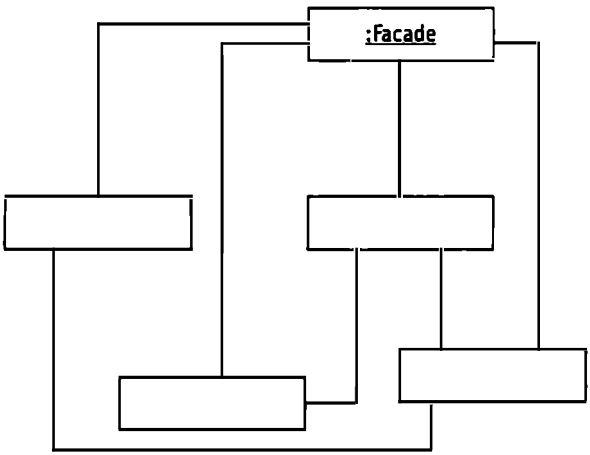


Рис. 2.19. Пустые объекты

Линии, соединяющие два объекта, не являются ассоциациями. Они называются *связями*. Связи соединяют объекты, в то время как ассоциации — это соотношения между классами. Связь представляет собой экземпляр ассоциации, точно

так же как объект — это экземпляр класса. Связи могут иметь имена ассоциаций, стрелки навигации и почти все остальные атрибуты, характерные для ассоциаций. Но поскольку связь представляет собой соединение двух объектов, то она не может иметь индикаторов множественности и ромбов агрегации.

Особым видом диаграмм класса считаются *диаграммы объектов*, содержащие только объекты и связи (рис. 2.20).

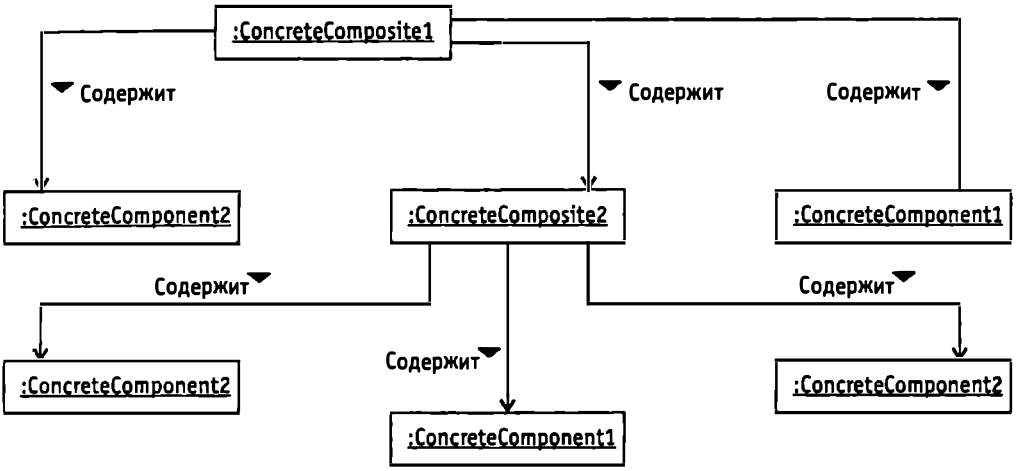


Рис. 2.20. Диаграмма объектов

Диаграмма взаимодействия

Диаграммы классов и объектов описывают соотношения между классами и объектами. Кроме того, они предоставляют информацию о взаимодействиях между классами. Они не показывают последовательность осуществления этих взаимодействий, а также возможную их параллельность.

Диаграммы взаимодействия описывают объекты, связи между ними и взаимодействия, осуществляемые через каждую связь. Они также показывают условия последовательности и параллельности для каждого взаимодействия (рис. 2.21).

С одной связью может ассоциироваться любое количество взаимодействий. Каждое взаимодействие предусматривает вызов метода. Рядом со взаимодействием или их группой располагается стрелка, указывающая на объект, метод которого вызывается. Полный набор объектов и взаимодействий представлен на диаграмме взаимодействия, которую обычно называют просто *взаимодействие*.

Обозначение каждого из представленных на рис. 2.21 взаимодействий начинается с порядкового номера и двоеточия. Порядковые номера описывают последовательность обращений к методам. Взаимодействие с номером 1 должно происходить перед взаимодействием с номером 2 и т.д.

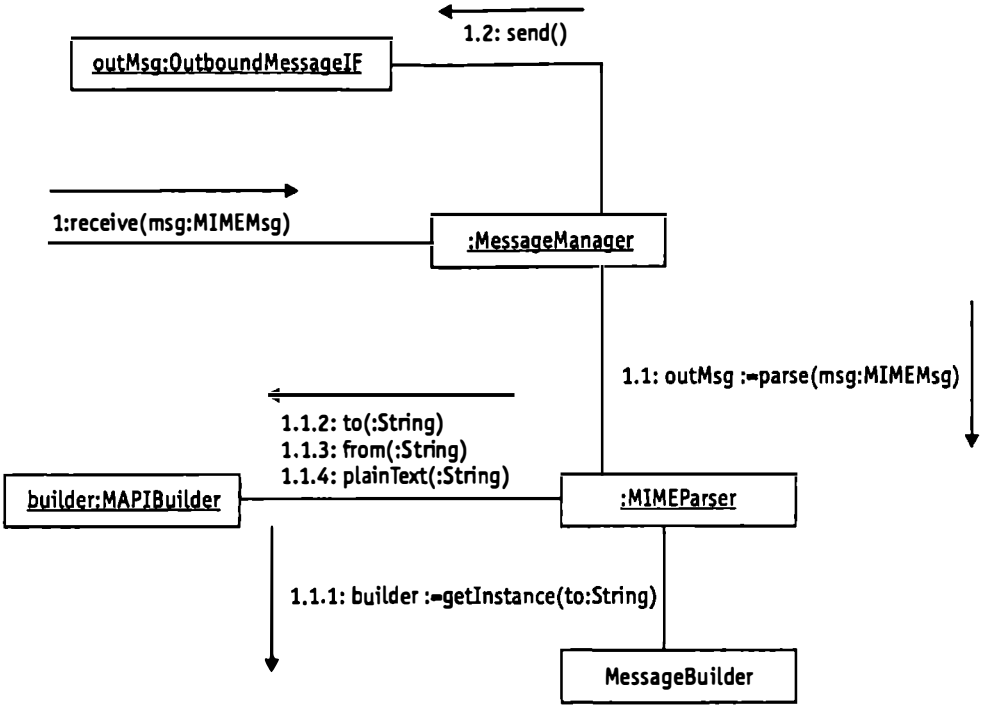


Рис. 2.21. Диаграмма взаимодействия

Составные порядковые номера включают в себя два и более числа, разделенные точкой. Такие номера соответствуют нескольким уровням обращений к методам. Часть составного порядкового номера, расположенная слева от самой правой точки, называется его *префиксом*. Например, префикс номера 1.3.4 — 1.3.

Взаимодействия, имеющие составной порядковый номер, происходят в то время, когда метод вызывается другим взаимодействием. Этот вызов метода задается префиксом взаимодействия. Таким образом, взаимодействия с номерами 1.1 и 1.2 вызывают методы в то время, когда происходит вызов метода взаимодействием номер 1. Аналогичным образом взаимодействия с номерами 1.1.1, 1.1.2, 1.1.3 и т.д. происходят во время вызова метода взаимодействием номер 1.1.

Во взаимодействиях с одинаковым префиксом последовательность вызова методов задается цифрой, следующей за префиксом. Поэтому методы взаимодействий с номерами 1.1.1, 1.1.2, 1.1.3 и т.д. вызываются именно в таком порядке.

Как было упомянуто ранее, связи представляют собой соединение двух объектов, поэтому связи не могут иметь индикаторов множественности. Связи удобно применять в тех случаях ассоциаций между объектами, когда количество объектов известно. Ассоциации с индикатором множественности в виде звездочки на одном из концов указывают на неограниченное количество объектов.

Для ассоциаций такого рода нельзя показать неопределенное количество связей, указывающих на неопределенное количество объектов. В UML для этого используется символ — мультиобъект. *Мультиобъект* представляет неограниченное число объектов. Он изображается в виде прямоугольника, находящегося позади другого прямоугольника (рис. 2.22).

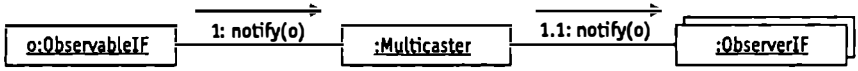


Рис. 2.22. Мультиобъект

На этой диаграмме взаимодействия изображен объект `ObservableIF`, вызывающий метод `notify` объекта `Multicaster`. Реализация метода `notify` объекта `Multicaster` вызывает метод `notify` у нескольких связанных с объектом `Multicaster` объектов `ObserverIF`, количество которых не определено.

Объекты, созданные в результате взаимодействия, могут помечаться свойством `{new}`. Временные объекты, которые существуют только во время взаимодействия, отмечены свойством `{transient`[1]`}`. На диаграмме взаимодействия, изображенной на рис. 2.23, представлено взаимодействие, создающее объект.

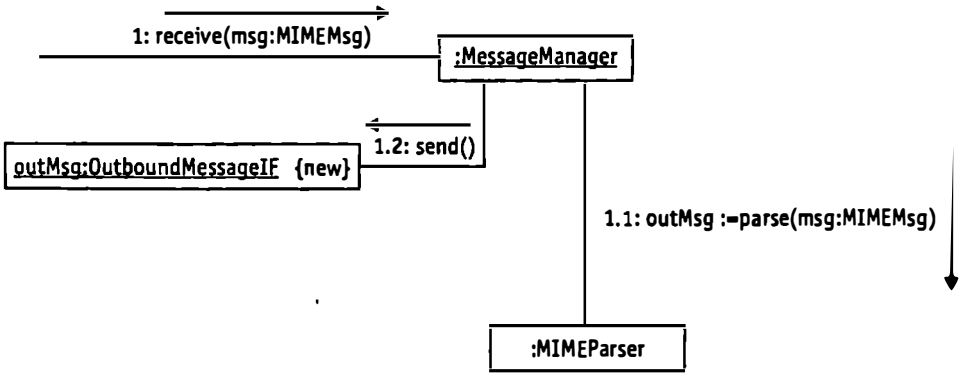


Рис. 2.23. Новый объект

Некоторые взаимодействия происходят не последовательно, а параллельно. Буква, расположенная в конце порядкового номера, указывает на параллельные взаимодействия. Например, методы взаимодействий с номерами 2.2a и 2.2b вызываются параллельно, и каждый вызов выполняется в отдельном потоке. Рассмотрим диаграмму взаимодействия (рис. 2.24).

---

[1] В UML и Java слово «transient» (временный) используется по-разному. В Java оно означает, что переменная не является частью постоянного состояния объекта. В UML этим словом обозначается объект, имеющий ограниченное время жизни.

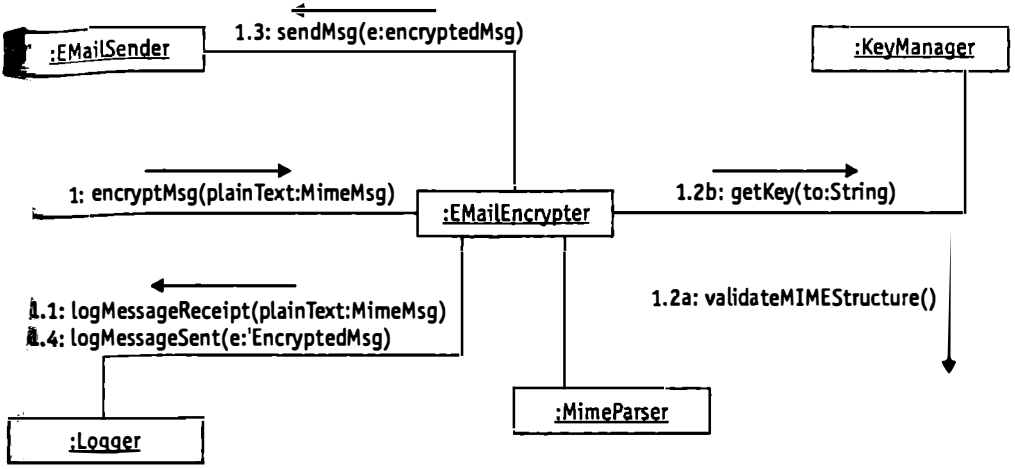
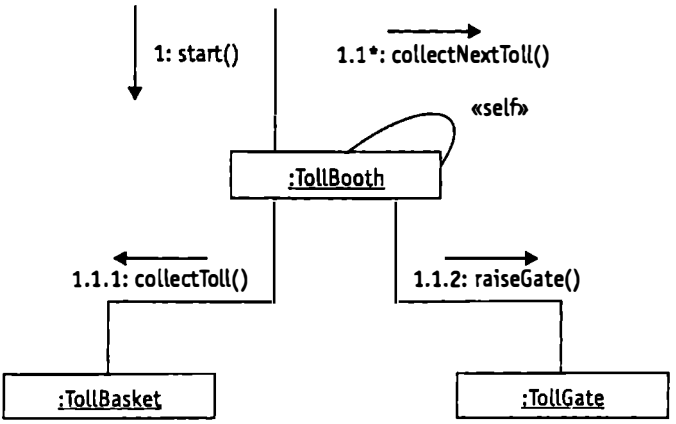


Рис. 2.24. Шифрование электронной почты

Отметим, что номер взаимодействия самого верхнего уровня — 1. Во время этого взаимодействия сначала активизируется взаимодействие 1.1. Затем одновременно активизируются взаимодействия 1.2a и 1.2b. Далее по порядку вызываются взаимодействия 1.3 и 1.4.

Звездочка, изображенная сразу за порядковым номером, указывает на повторяющееся взаимодействие.

Взаимодействие, изображенное на рис. 2.25, начинается с вызова метода `start` объекта `TollBooth`. Этот метод выполняет повторяющиеся вызовы метода `collectNextToll` объекта. Каждое обращение к методу `collectNextToll` вызывает метод `collectToll` объекта `TollBasket` и метод `raiseGate` объекта `TollGate`.



При рассмотрении данной диаграммы взаимодействия следует отметить еще один момент — стереотип «self», расположенный рядом со связью взаимодействия 1.1. Данный стереотип указывает на то, что связь является ссылкой на сам объект.

В отличие от примера на рис. 2.25 большинство повторяющихся взаимодействий осуществляется при выполнении некоторых условий. UML позволяет указывать такие условия в квадратных скобках перед двоеточием. На рис. 2.26 показан пример условного повторяющегося взаимодействия, в ходе которого объект `Iterator` передается методу `refresh` объекта `DialogMediator`. Его метод `refresh`, в свою очередь, вызывает метод `reset` объекта `Widget`, а затем периодически вызывает метод `addData` этого объекта до тех пор, пока метод `hasNext` объекта `Iterator` возвращает true.

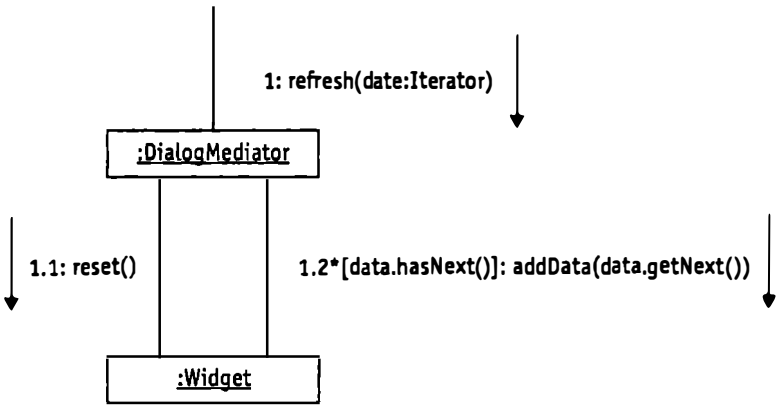


Рис. 2.26. Обновление

Обратите внимание, что UML не задает точное значение условий, связанных с повторяющимися взаимодействиями. Спецификация UML сообщает, например, что находящаяся в квадратных скобках информация может «быть выражена при помощи псевдокода или реального языка программирования». В данной книге для этой цели везде используется язык программирования Java.

При рассмотрении многопоточности часто нужно учитывать ситуацию, когда два потока пытаются одновременно вызвать один и тот же метод. UML предполагает в таком случае наличие одного из следующих выражений, указанного после метода:

```
{concurrency = sequential}
```

Это значит, что в какой-то момент времени только один поток должен вызывать метод. Никто не гарантирует правильное поведение метода в том случае, если в определенный момент времени метод вызывается несколькими потоками одновременно.

```
{concurrency = concurrent}
```

Это означает, что если несколько потоков вызывают метод одновременно, то они все выполняют его параллельно и правильно.

```
{concurrency = guarded}
```

Это значит, что если несколько потоков вызывают метод одновременно, то в какой-то момент времени только одному потоку будет разрешено выполнять метод. Пока один поток выполняет метод, другие потоки вынуждены ждать своей очереди. Это похоже на поведение синхронизированных методов в Java.

Диаграмма взаимодействия, представленная на рис. 2.27, демонстрирует пример синхронизированного метода.

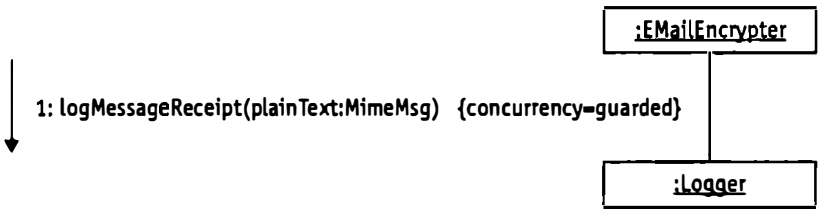


Рис. 2.27. Вызов синхронизированного метода

Существуют определенные тонкости при синхронизации потоков, рассмотренные в данной книге и не имеющие стандартного представления в UML. Для описания этих тонкостей в конструкции `{concurrency = guarded}` используются дополнительные пояснения.

В некоторых случаях объект, потоки которого должны быть синхронизированы, отличается от объекта, метод которого вызывается взаимодействием с этими потоками. Рассмотрим рис. 2.28. Представленное на этой диаграмме выражение `{concurrency = guarded:lockObject}` ссылается на объект `lockObject`. Перед реальным обращением к методу поток, контролирующий вызов, должен заблокировать объект `out`. В семантике Java это соответствует синхронизированному оператору.

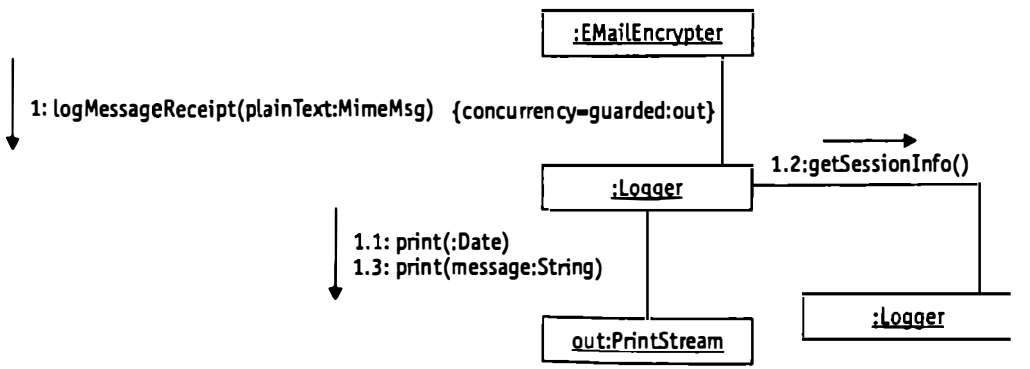


Рис. 2.28. Синхронизация, использующая третий объект

Иногда кроме блокировки необходимы некоторые другие предварительные условия, которые должны удовлетворяться до того, как поток сможет осуществить вызов метода. В данной книге перед такими условиями стоит вертикальная черта. На диаграмме взаимодействия, представленной на рис. 2.29, указаны предварительные условия (им предшествует слово `guarded` и вертикальная черта).

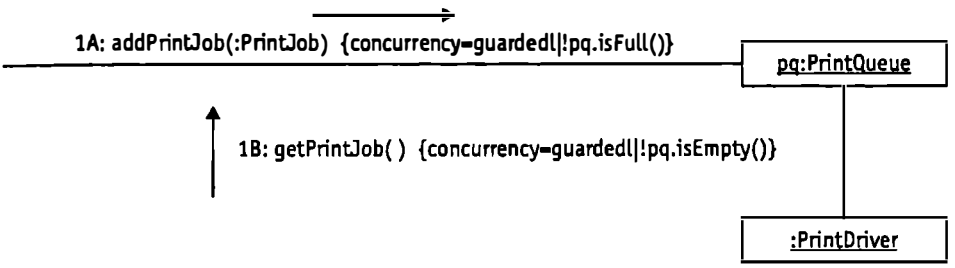


Рис. 2.29. Очередь заданий вывода на печать

Эта диаграмма взаимодействия описывает два асинхронных взаимодействия. Одно вызывает метод `addPrintJob` объекта `PrintQueue` с целью добавления задания по выводу на печать в объект `PrintQueue`. В ходе других взаимодействий объект `PrintDriver` вызывает метод `getPrintJob` объекта `PrintQueue` для извлечения задания по выводу на печать из объекта `PrintQueue`. Оба взаимодействия имеют предварительные условия синхронизации. Если очередь принтера заполнена, то вызывающее метод `addPrintJob` взаимодействие ожидает свободного места в очереди, перед тем как вызвать метод `addPrintJob`. Если очередь принтера пуста, то перед тем, как вызвать метод `getPrintJob`, вызывающее этот метод взаимодействие ожидает, пока в очереди не появится задание.

Существуют такие предварительные условия, которые, как правило, задаются не с помощью выражения, а требуют, чтобы некоторое взаимодействие не начиналось до тех пор, пока не закончатся два или более других взаимодействий. Неявное предварительное условие для всех взаимодействий состоит в том, что они не начинаются до тех пор, пока не закончится взаимодействие, непосредственно им предшествующее. Например, взаимодействие с номером 1.2.4 не начнется до тех пор, пока не завершится взаимодействие 1.2.3.

Перед тем как начаться, некоторые взаимодействия должны ожидать завершения дополнительных, или предварительных, взаимодействий. Такие дополнительные (предварительные) взаимодействия представлены в виде списка, указанного слева от взаимодействия; затем идет наклонная черта (/) и остальная часть описания взаимодействия. На диаграмме взаимодействия, изображенной на рис. 2.30, показан пример предварительных взаимодействий.

Согласно рис. 2.30, взаимодействие 2.1a.1 не должно начинаться до тех пор, пока не закончится взаимодействие 1.1.2. Если перед тем, как начаться, взаи-

модействие должно ожидать окончания не одного, а нескольких дополнительных предварительных взаимодействий, они указываются перед наклонной чертой и отделяются друг от друга запятыми.

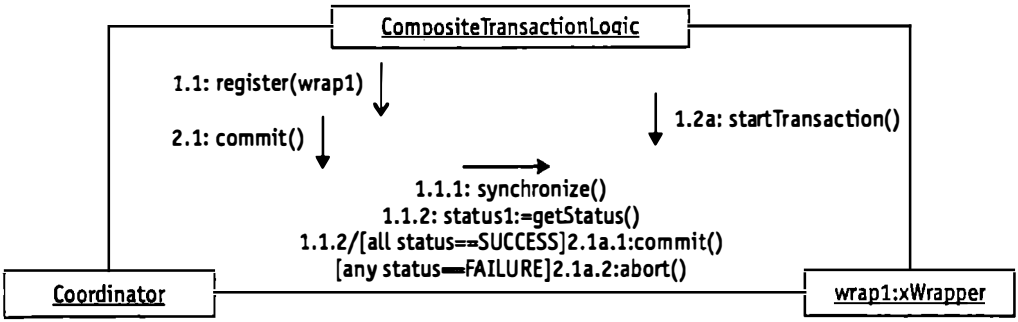


Рис. 2.30. Дополнительные (предварительные) взаимодействия

Механизмы, рассматривавшиеся до настоящего момента, определяют, когда вызываются методы взаимодействия. Однако они ничего не сообщают о том, когда методы завершают свою работу. Стрелки, которые указывают на объекты, содержащие вызываемые методы, информируют о том, когда эти методы могут закончить свою работу.

Почти все стрелки, изображенные на рис. 2.30, заканчиваются замкнутым контуром, что свидетельствует о синхронности этих вызовов. Выполнение метода не осуществляется до тех пор, пока метод не сделает все то, что он должен сделать.

Незамкнутые контуры стрелок указывают на асинхронный вызов метода, который выполняется сразу. Метод в это время работает асинхронно, в отдельном потоке. Изображенная на рис. 2.31 диаграмма взаимодействия демонстрирует асинхронный вызов метода.

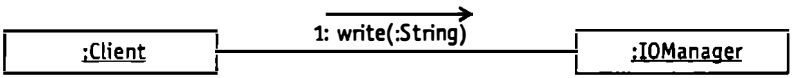


Рис. 2.31. Асинхронный вызов метода

UML определяет вид стрелок только для синхронных и асинхронных обращений. Для указания других различных обращений к методам допускается применение расширения UML, которое предусматривает использование стрелок других видов. Отменяемый вызов в этой книге обозначается стрелкой, изогнутой в обратном направлении (рис. 2.32).

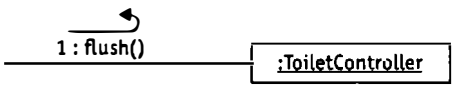


Рис. 2.32. Отменяемый вызов

Если при отменяемом вызове метода объекта никакие другие потоки не выполняют методы этого объекта, то он заканчивается, выполнив свою работу. Однако если при отменяемом вызове метода существует другой поток, выполняющий в данный момент этот метод объекта, то он завершается сразу, не выполнив никаких действий.

Вы могли заметить, что объект, который осуществляет вызов высшего уровня, инициирующий сотрудничество, не показан ни на одной диаграмме взаимодействия. Если объект, инициирующий сотрудничество, не указан на диаграмме взаимодействия, то этот объект не считается частью взаимодействия.

До этого мы рассматривали описываемые с помощью UML объекты, пассивные по своей природе. Они ничего не делают до тех пор, пока не вызывается один из методов этих объектов.

Некоторые объекты являются активными. Они связаны с потоком, позволяющим им инициировать операции асинхронно и независимо от любых действий программы. Активный объект изображается в виде прямоугольника с более жирным контуром (рис. 2.33).

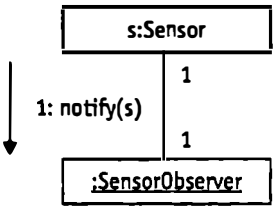


Рис. 2.33. Активный объект Sensor

На диаграмме показан активный объект `Sensor`, вызывающий метод объекта `SensorObserver`, и этому не предшествует обращение другого объекта к одному из методов этого объекта.

Диаграмма состояний

Диаграммы состояний (рис. 2.34) предназначены для моделирования поведения класса как конечного автомата.

На диаграмме состояний каждое состояние изображено в виде прямоугольника с закругленными углами. Все состояния, показанные на рис. 2.34, разделены на две части. Верхняя часть содержит имя состояния, нижняя — список событий, на которые реагирует объект, находясь в данном состоянии и не изменяя это состояние. За каждым событием списка следует наклонная черта и описание действия, выполняемого в ответ на событие. UML предопределяет два события такого рода:

- событие `enter` происходит тогда, когда объект входит в состояние;
- событие `exit` происходит тогда, когда объект выходит из состояния.

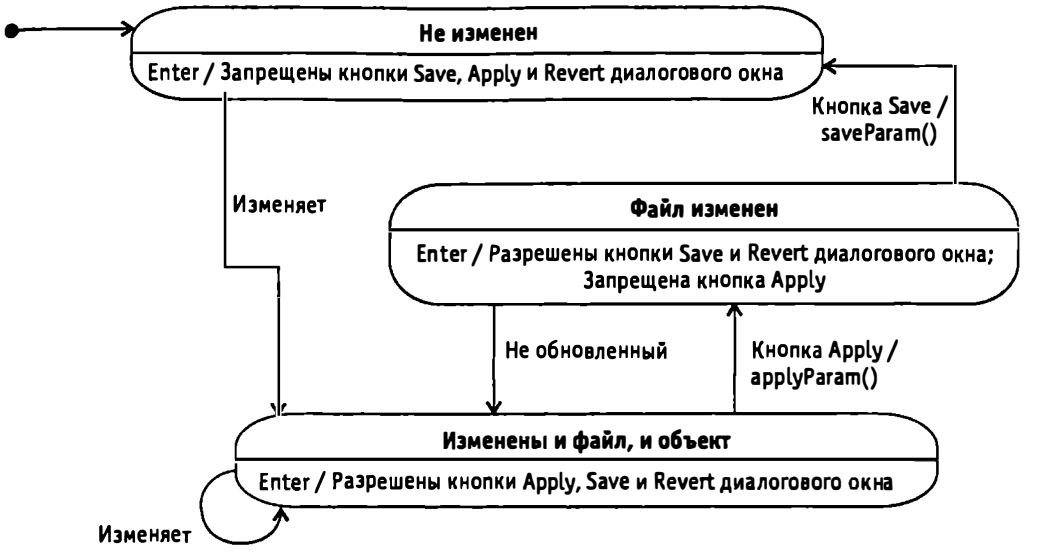


Рис. 2.34. Диаграмма состояний

Если все события заставляют объект изменять свое состояние, прямоугольник состояния не разделяется на две части. При этом прямоугольник с закругленными углами содержит только имя состояния.

Каждый конечный автомат имеет начальное состояние, в котором находится объект перед первым переходом в другое состояние. Исходное состояние изображается в виде маленького закрашенного кружка.

На диаграмме состояний переходы между состояниями показаны в виде линий. Обычно линия перехода должна иметь метку, обозначающую событие, которое инициирует переход. За событием может следовать наклонная черта и описание действия, выполняемого во время перехода.

Если диаграмма содержит конечное состояние, то оно изображается в виде маленького закрашенного кружка, находящегося внутри более крупного кружка.

Диаграмма развертывания

Диаграмма развертывания показывает, каким образом ПО развертывается на вычислительные элементы. На рис. 2.35 представлен пример диаграммы развертывания.

На рис. 2.35 изображены два вычислительных элемента, помеченных как `Server1` и `Server2`. Согласно терминологии UML, вычислительный элемент — это *узел*. Но так как в данной книге это слово используется для обозначения других вещей, все-таки будем использовать термин *вычислительный элемент*.

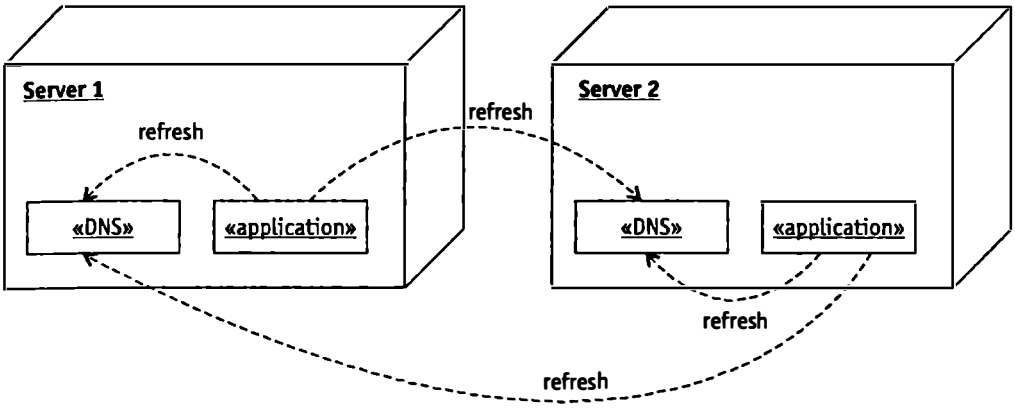


Рис. 2.35. Диаграмма развертывания

Прямоугольники внутри вычислительных элементов представляют собой компоненты ПО. Компонент — это коллекция объектов, которые развертываются все вместе. На этой диаграмме каждый вычислительный элемент имеет два компонента: компонент `DNS` и компонент `application`.

Связи между компонентами обозначены пунктирными линиями. Как показано на рис. 2.35, пунктирные линии, помеченные словом `refresh`, свидетельствуют о том, что компоненты `application` отправляют сообщения компонентам `DNS`.

ГЛАВА 3

Жизненный цикл программного обеспечения

Эта книга посвящена шаблонам, используемым на стадии объектно-ориентированного проектирования, которая является частью жизненного цикла ПО. В этой главе сначала описывается жизненный цикл, а затем представлена его часть, связанная с объектно-ориентированным проектированием, выполняемым при исследовании проблемы.

За время жизни программного продукта предпринимаются разнообразные действия.

На рис. 3.1 показана часть всех действий по развертыванию ПО для бизнеса, выполняемых на этапе проектирования программного продукта. На нем представлены только некоторые общие действия, которые выполняются с целью выяснения контекста использования шаблонов, рассматриваемых в книге. В книге описываются повторяющиеся шаблоны, используемые на протяжении той части жизненного цикла ПО, которая на рис. 3.1 помечена как «Построение».

Рис. 3.1 демонстрирует очень четкие границы между действиями. В действительности границы не всегда являются такими точными. Иногда трудно определить, к какому этапу принадлежит определенное действие. Не так важна точность границ, как понимание отношений между этими действиями.

Усилия, предпринимаемые на начальных этапах (например, определение требований или объектно-ориентированный анализ), задают направление последующих действий, например, определяют ключевые ситуации использования или объектно-ориентированное проектирование. Однако на этих более поздних этапах возникает дефицит продуктов, полученных на ранних стадиях. При определении ситуации использования может оказаться, например, что выдвигается неоднозначное или несовместимое требование. Необходимость изменения требований приводит, главным образом, к необходимости изменения существующих ситуаций использования или написания новых. Такие итерации вполне вероятны. Итерации, выполняемые на более поздних этапах, предполагают меньше изменений, чем итерации, реализуемые на более ранних стадиях.

Опишем некоторые действия, представленные на рис. 3.1. Описание прост предоставляет пользователю достаточную информацию об этих действиях, чтобы он понимал, каким образом описываемые в данной книге шаблоны могут применяться в рамках соответствующего действия. Исследование проблемы, следующее за описанием, подразумевает более глубокое рассмотрение действий.

Рис. 3.1. Действия, приводящие к развертыванию ПО

Бизнес-планирование. Как правило, оно начинается с предложения создать или изменить какую-либо часть ПО. Предложение входит в бизнес-план. *Бизнес-план* — это документ, который описывает все «за» и «против» проекта программного продукта и, кроме того, содержит оценку ресурсов, необходимых для выполнения проекта. Если принимается решение приступить к проектированию, то подготавливается предварительный план и бюджет.

Определение требований. Целью этого действия является создание *спецификации требований*, в которой указано, что будет и что не будет делать проектируемый программный продукт. Как правило, сначала на основании бизнес-плана определяются цели и требования высокого уровня. Для создания начальной спецификации требований из соответствующих источников берутся дополнительные требования.

По мере использования спецификации для последующих действий, возникает необходимость уточнения требований. Уточнения вносятся в спецификацию требований. Затем изменяются результаты последующих действий с целью

Определение ситуаций использования. *Ситуация использования* описывает последовательность событий, происходящих между системой и другими объектами (*исполнителями* — actors) при определенных обстоятельствах. В результате разработки ситуаций использования появляется более глубокое представление о требованиях, анализе или проектировании, составляющих основу ситуации использования.

Ключевые ситуации использования описывают события с точки зрения проблемной области. Ситуации использования, описывающие события с точки зрения внутренней структуры программного продукта, называются *реальными ситуациями использования.*

Ситуации использования, более всего приспособленные для уточнения требований, представляют собой *ключевые ситуации использования высокого уровня.* Высокий уровень таких ситуаций означает, что они исследуют принципы, на которых они основаны, и не пытаются добавлять дополнительные детали.

Создание прототипа. Целью этого действия является создание прототипа будущего программного продукта, который может применяться для получения откликов на готовяшийся проект. Результаты прототипирования могут быть использованы для уточнения требований и ситуаций использования.

Следующие два этапа относятся к определению архитектуры системы высокого уровня. Цель этого действия заключается в определении основных компонентов системы, которые являются логическим следствием первоначальных предположений и их отношений.

Объектно-ориентированный анализ. Цель данного действия — выяснить, что будет делать описанный в проекте программный продукт и как он будет взаимодействовать с другими объектами своего окружения. В ходе анализа должна быть создана модель того, ЧТО будет делать ПО, но не КАК оно это будет делать. Продукты объектно-ориентированного анализа моделируют ситуацию, в рамках которой будет работать программный продукт, с точки зрения внешнего наблюдателя. Сам по себе анализ не затрагивает всего происходящего внутри программного продукта.

Объектно-ориентированное проектирование. Целью данного действия является определение внутренней структуры ПО. Продукты проектирования идентифицируют классы, образующие внутреннюю логику ПО. Они задают также внутреннюю структуру классов и их взаимоотношения.

На этапе объектно-ориентированного проектирования принимается больше решений, чем на любом другом. Поэтому в данной книге уделяется особое внимание шаблонам, которые более всего подходят для объектно-ориентированного проектирования, чем для любого другого вида деятельности.

Кодирование. На этом этапе пишется код, заставляющий ПО работать.

Тестирование. Результатом этого этапа является гарантия выполнения программным продуктом своих функций в соответствии с поставленными задачами.

Исследование проблемы

Рассмотрим ситуацию использования, которая включает проектирование и разработку системы учета рабочего времени служащих выдуманной компании Henry's Food Market («Продовольственная сеть Генри»). Некоторые обстоятельства процесса разработки упрощены и сокращены. Задача данного исследования состоит в определении места действия ситуации, демонстрирующего применение шаблонов проектирования.

Бизнес-планирование

Здесь представлено упрощенное бизнес-планирование, описывающее мотивацию и планирование, связанные с созданием системы учета рабочего времени служащих.

«Продовольственная сеть Генри» состоит из пяти магазинов, товарного склада и булочной, которая производит выпечку, реализуемую через магазины. Большинство служащих компании получает почасовую оплату. Рабочее время сотрудников отслеживается системой учета времени. Приходя на работу, выходя на перерыв, возвращаясь с перерыва или уходя с работы, все служащие должны пропускать свои идентификационные карточки через устройство учета времени, которое фиксирует рабочее время.

«Продовольственная сеть Генри» намерена расшириться и в течение двух лет увеличить количество своих магазинов с 5 до 21, открывая каждые три месяца по два новых магазина. При этом возникает проблема: если компания Генри продолжает эксплуатировать прежнюю систему учета времени, то она должна будет нанять большее количество людей для администрирования системы учета времени. На данный момент каждую торговую точку должен обслуживать сотрудник, который половину смены регистрирует рабочее время и обслуживает систему учета времени этой точки. В функции регистратора входят следующие действия.

- Он печатает отчеты, содержащие количество часов занятости каждого сотрудника на работе в течение последнего дня. Это позволяет контролерам убедиться в том, что их подчиненные работали установленное количество часов. Контролеры, просматриваюшие такие отчеты, чаще всего обнаруживают следующие нарушения:

служащие, уходящие на перерыв или с работы, не отметили время ухода;

коллеги отметили время прихода служащих, когда те опоздали на работу;

служащие отметили время прихода до начала своей рабочей смены, надеясь на оплату сверхнормативного времени.

- Регистратор вносит изменения в систему учета рабочего времени.
- Регистратор готовит еженедельные отчеты, в которых указано, сколько часов проработал каждый служащий, и отправляет эти отчеты в отдел платеж-

Используемая система учета времени предоставляет данные по количеству отработанных часов только в виде распечатанного отчета. В настоящее время эту работу в течение всего рабочего дня выполняет один человек, который вводит количество отработанных служащими часов в систему платежных ведомостей и просматривает введенную информацию. Этот работник «стоит» компании $ 24 000 в год. Если компания продолжит использование этой системы, она должна будет платить $ 24 000 в год еще одному человеку, который будет вводить данные рабочего времени служащих.

Стоимость найма человека, частично занятого в должности регистратора в каждом магазине, составляет $ 9000 на 1 работника в год. Текущая стоимость содержания штата регистраторов — $ 63 000 в год.

Полная текущая стоимость затрат на поддержание системы учета времени составляет $ 87 000 в год. Через два года, после расширения компании, стоимость этой работы возрастет до $ 237 000.

Будущий проект должен создать новую систему учета рабочего времени, которая позволит и после расширения сети оставить на прежнем или более низком уровне стоимость работ по поддержанию системы учета времени. Ожидается, что эта система окупит себя в течение 18 месяцев и на ее развертывание потребуется 6 месяцев с момента начала проекта.

Определение спецификации требований

Как минимум, спецификация требований должна задавать необходимые функции и атрибуты всего, что присутствует в проекте. *Необходимые функции* — это все действия, которые система должна делать (например, регистрация времени начала работы служащих). *Необходимые атрибуты* — это характеристики системы, не являющиеся функциями (например, условие, в соответствии с которым терминалами регистрации могут пользоваться люди, получившие хотя бы базовое образование). Существуют также некоторые другие вещи, которые могут присутствовать в спецификации требований, но не упоминаются в этой книге.

Предположения. Это перечень справедливых утверждений, таких как требование минимального образовательного уровня служащих или то, что компания не будет объединена.

Риски. Представляют собой перечень вероятных ошибочных процессов, которые приводят к задержке или несостоятельности работы проекта. Такой список может содержать сомнительные технические аспекты (например, доступность устройств, пригодных для использования в качестве терминалов учета времени). Кроме того, список может включать пункты, не связанные с техникой (например, предполагаемые изменения в законе о труде).

Зависимости. Представляют собой перечень ресурсов, от которых может зависеть данный проект (например, существование локальной сети).

Требования, указанные в спецификации, лучше пронумеровать. Тогда основан-

использования, документах проекта и даже исходном тексте программы. Если позднее обнаружатся несоответствия, то несложно отследить их в обратном направлении и установить связь с определенными требованиями. Обычно требования нумеруются в иерархическом порядке с использованием функций. Опишем некоторые функции, необходимые для системы учета времени.

R1. Система должна документировать время, когда служащие начинают работу, идут на перерыв, возвращаются с перерыва и уходят с работы.

R1.1. При использовании терминала учета рабочего времени служащие должны идентифицировать себя, помещая свою идентификационную карточку в устройство для считывания на терминале.

R1.2. После идентификации с помощью терминала учета рабочего времени служащий может нажать кнопку, которая отмечает либо начало рабочей смены, либо уход на перерыв, либо возвращение с перерыва, либо конец смены. Система учета времени долговременно хранит запись каждого такого события в форме, позволяющей помещать ее в отчет, документирующий рабочие часы служащих.

R2. Контролеры должны иметь возможность просматривать рабочее время подчиненных на терминале, не распечатывая эти данные.

R2.1. Терминал учета времени позволяет контролеру просматривать и видоизменять зарегистрированные рабочие часы служащего.

R2.1.1. Все выполненные ревизии записи учета рабочего времени служащих отражаются в контрольном журнале, где сохраняются также исходные записи и в каждом случае указывается имя контролера, проводившего ревизию.

R2.2. Пользовательский интерфейс терминала для работников, не являющихся контролерами, не должен содержать меню, связанные с функциями контролеров.

R2.3. Контролеры могут изменять записи учета рабочего времени только своих подчиненных.

R3. По завершении каждого платежного периода система учета рабочего времени должна автоматически передавать данные о рабочих часах сотрудника системе платежных ведомостей.

По мере разработки некоторых ситуаций использования возможно появление новых полезных функций.

Разработка ключевых ситуаций использования высокого уровня

При разработке ситуаций использования сначала, как правило, основное внимание уделяется наиболее распространенным случаям, а затем разрабатываются менее используемые ситуации. Широко распространенные ситуации использо-

вания называются *основными ситуациями использования*. Реже встречающиеся ситуации называются *второстепенными ситуациями использования*. Опишем ситуацию использования для часто встречающегося случая применения системы учета рабочего времени.

Ситуация использования:	Сотрудник использует терминал учета времени.
Участник:	Сотрудник.
Цель:	Ввод информации об уходе и приходе сотрудника в систему учета рабочего времени.
Синопсис:	Сотрудник собирается начать работу, уйти на перерыв, вернуться с перерыва или закончить рабочую смену. Он идентифицирует себя в системе учета времени и дает ей знать, какой из четырех вариантов он собирается осуществить.
Тип:	Основной и ключевой.
Перекрестные ссылки:	Требования R1, R1.1, R1.2, R1.3 и R2.2.

Ход событий

Сотрудник	Система
1. Сотрудник проводит своей идентификационной карточкой через считывающее устройство терминала	2. Терминал учета времени считывает идентификационный номер сотрудника с карточки и проверяет правильность этого номера. Затем терминал учета «напоминает» сотруднику, что он должен указать: или начало работы, или уход на перерыв, или возвращение с перерыва, или конец рабочей смены
3. Сотрудник указывает на терминале, что он либо начинает работу, либо уходит на перерыв, либо приходит с перерыва, либо заканчивает рабочую смену	4. Терминал учета рабочего времени делает долгосрочную запись данных сотрудника. Затем подтверждается правильность данных и отображается текущее время. Это свидетельствует о готовности приема данных от следующего сотрудника

Рассмотрим менее детализированную ситуацию использования.

Ситуация использования:	Сотрудник использует несколько терминалов учета времени для отслеживания рабочих часов.
Участник:	Сотрудник.
Цель:	Ввод в систему учета информации об уходе и приходе сотрудника в течение всей рабочей смены.

Синопсис:	Сотрудник может выбрать любой терминал учета времени, чтобы сообщить системе: начинает он работу, уходит на перерыв, возвращается с перерыва или заканчивает рабочую смену.
Тип:	Основной и ключевой.
Перекрестные ссылки:	Требования R1 и R1.2.

Ход событий

Сотрудник	Система
1. Сотрудник использует терминал учета рабочего времени, чтобы проинформировать систему о начале работы	2. Система записывает время начала рабочей смены служащего
3. Сотрудник использует терминал для информирования системы о том, что он идет на перерыв	4. Система записывает время ухода служащего на перерыв
5. Сотрудник использует терминал для информирования системы о том, что он возвращается с перерыва	6. Система записывает время возвращения служащего с перерыва
7. Сотрудник использует терминал для информирования системы об окончании работы	8. Система записывает время окончания сотрудником рабочей смены

При анализе менее детализированной ситуации использования может возникнуть проблема. Не существует условия, которое бы гласило, что все терминалы учета рабочего времени должны показывать точное время. Скорее всего, сотрудники заметят неодинаковое отображение времени на разных терминалах. Им захочется начинать свою рабочую смену с терминала, показывающего более раннее время, и, заканчивая работу, проходить через терминал, показывающий более позднее время. Чтобы сотрудники не имели такой возможности обмана компании, необходимо добавить еще одно требование:

R1.3. Разность показаний времени, отображаемых и записываемых разными терминалами, не должна превышать пять секунд.

Как только будут разработаны ключевые случаи ситуации использования, будут найдены новые уточнения требований.

Объектно-ориентированный анализ

Объектно-ориентированный анализ занимается построением модели проблемы, подлежащей решению. Он отвечает на вопрос, что будет делать программ-

Основным результатом объектно-ориентированного анализа является концептуальная модель проблемы, описывающая предлагаемую систему и объекты реального мира, с которыми система будет взаимодействовать. Кроме того, концептуальная модель рассматривает отношения и взаимодействия между объектами проблемной области, а также между объектами и системой.

Как правило, концептуальные модели создаются в два этапа:

1. Идентификация объектов, охватываемых проблемой. Очень важно обозначить все относящиеся к проблеме объекты. Если есть сомнения, то лучше включить объект в модель. Если объект не нужен для последующего проектирования, это становится очевидным во время разработки проекта. С другой стороны, если нужный объект отсутствует при анализе, то его отсутствие может быть не обнаружено позднее в проекте.
2. Создание концептуальной модели для идентификации отношений между объектами.

Для описания объектов и отношений концептуальной модели UML использует ту же символику, которую он применяет при описании классов и ассоциаций в модели классов. На рис. 3.2 изображена диаграмма, содержащая (в произвольном порядке) только те объекты, присутствие которых обусловлено требованиями и ситуациями использования. Изображенная на рис. 3.3 диаграмма содержит наиболее очевидные отношения.

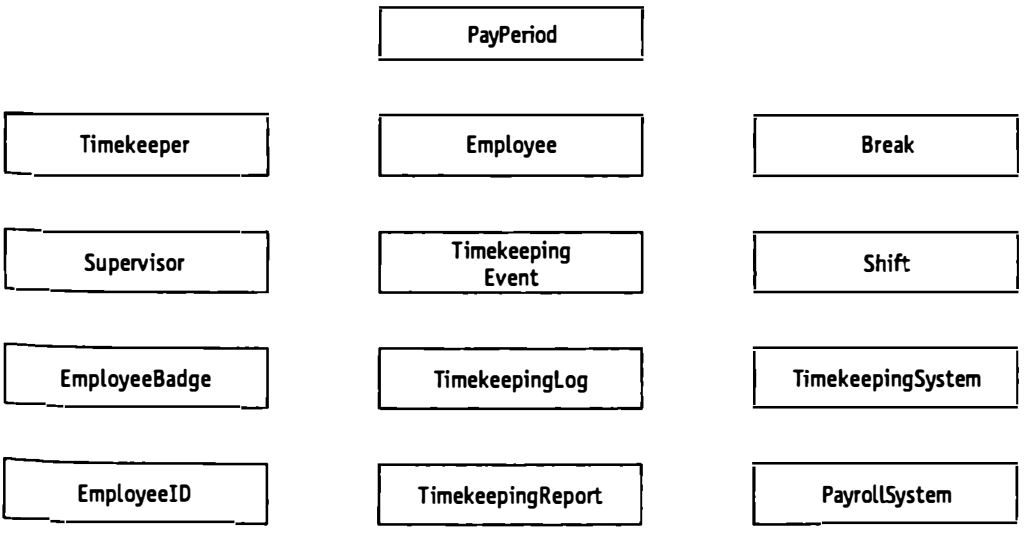


Рис. 3.2. Концептуальная модель, содержащая только объекты

При внимательном рассмотрении рис. 3.3 можно обнаружить два объекта, которые не участвуют ни в одном из указанных отношений, это объекты `TimekeepingSystem` и `EmployeeID`.

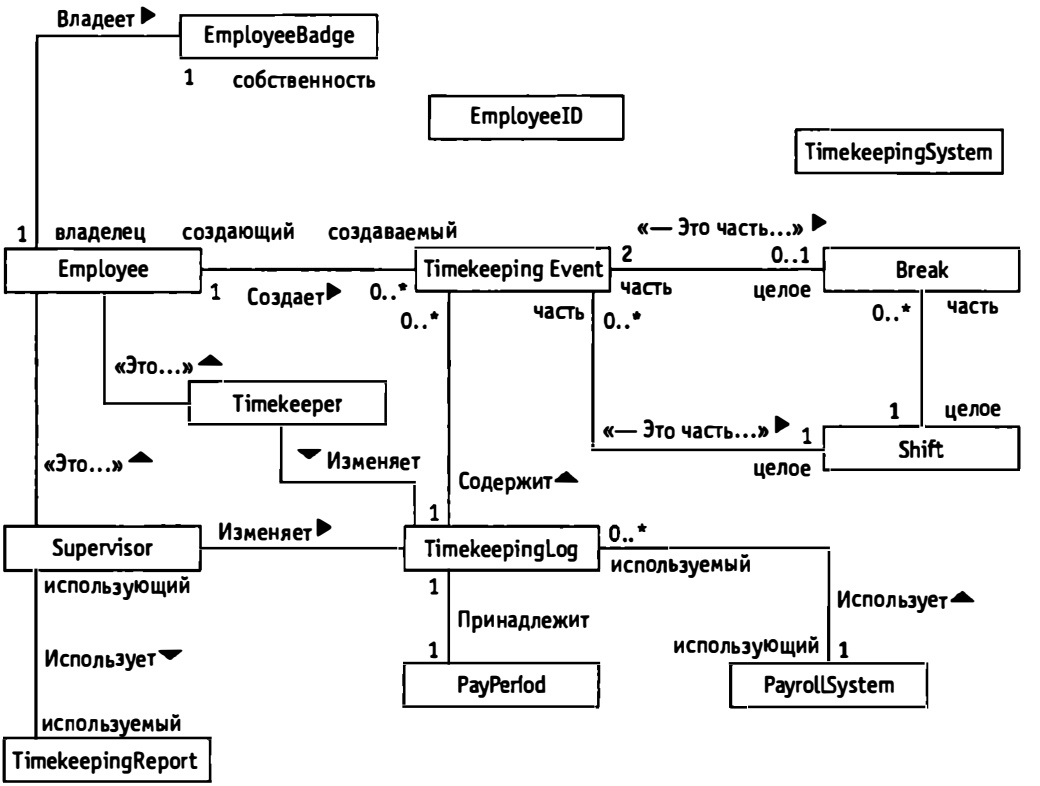


Рис. 3.3. Концептуальная модель, представленная вместе с ассоциациями

Данная диаграмма считается концептуальной моделью искомой проблемы. Как оказалось, объект `TimekeepingSystem` в рамках проблемы ни с чем не связан, поэтому делаем вывод, что он является частью решения, а не проблемы. Значит, его можно удалить из модели.

Объект `EmployeeID` очень тесно связан с объектом `Employee`. В таком случае его лучше представлять в виде атрибута. На рис. 3.4 представлена концептуальная модель, в которую добавлены атрибуты.

Эта диаграмма отражает всю глубину проведенного нами анализа данной проблемы.

Объектно-ориентированное проектирование

Объектно-ориентированное проектирование занимается разработкой внутренней логики программы — в данном случае рассматривается внутренняя логика системы учета рабочего времени. Проект не учитывает ни то, как пользователь-

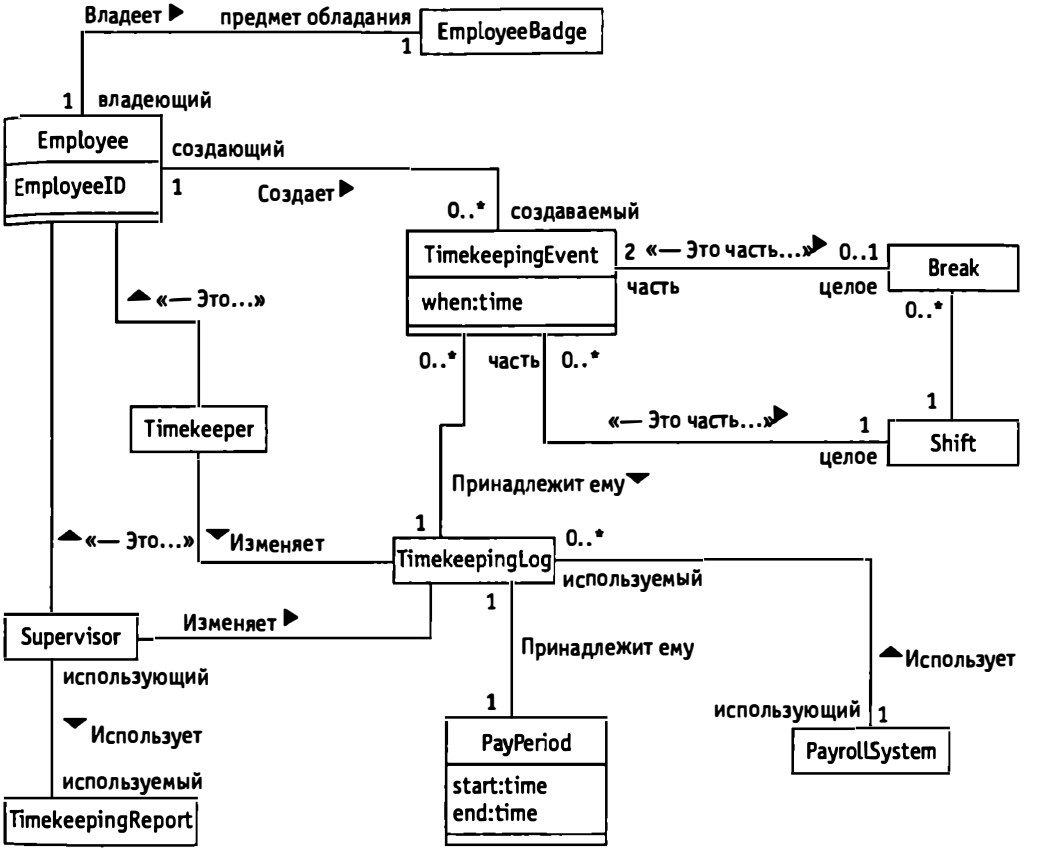


Рис. 3.4. Концептуальная модель с атрибутами

данных. Конечной целью объектно-ориентированного проектирования является создание детального проекта классов, обеспечивающих эту внутреннюю логику.

Существуют различные стратегии использования результатов анализа при создании проекта. Стратегия, используемая в данной книге, заключается в создании диаграммы классов, моделирующей структуру отношений в концептуальной модели. Затем разрабатываются диаграммы взаимодействия с целью моделирования поведенческих отношений в рамках концептуальной модели. Далее уточняются диаграммы классов с учетом информации, предоставляемой диаграммами взаимодействия. И наконец, диаграммы классов и диаграммы взаимодействия дополняются требованиями, которые не были охвачены концептуальной моделью. В течение всего процесса используются шаблоны проектирования, способствующие достижению цели.

Создадим первую диаграмму классов, предполагая, что существует класс для представления каждого объекта концептуальной модели (рис. 3.5). Данная диаграмма не представляет атрибутов объектов концептуальной модели, а описы-

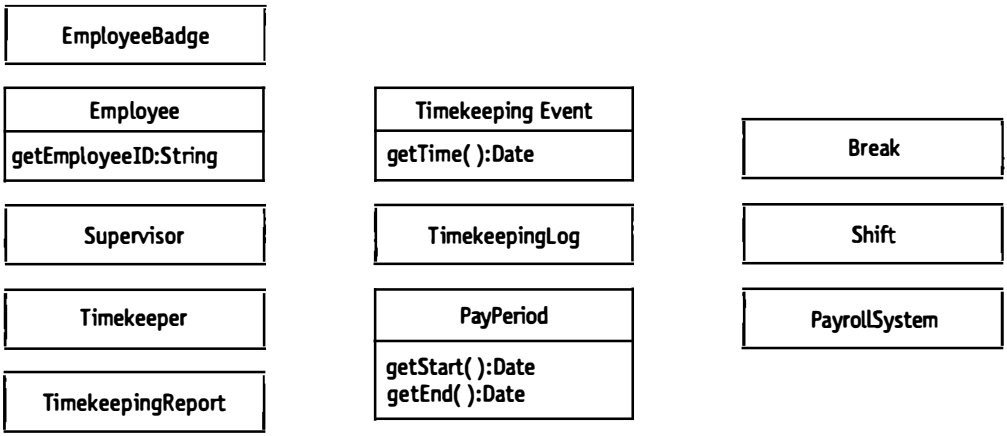
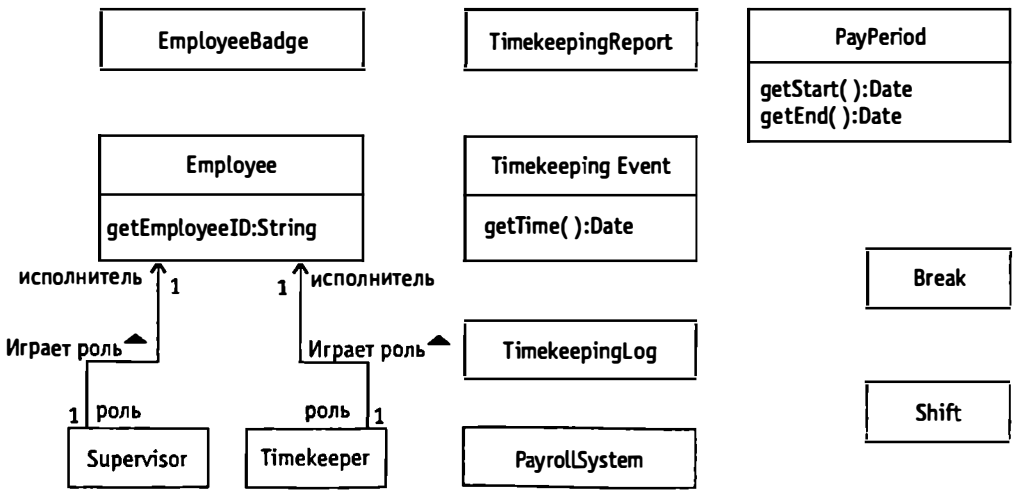


Рис. 3.5. Диаграмма классов, версия 1

Рассмотрим описываемые в концептуальной модели отношения «— Это...» («Is-a»). Хотя очевидным способом представления отношения «— Это...» на диаграмме классов является наследование, однако шаблон Delegation свидетельствует о том, что не всегда этот способ является наилучшим при представлении отношений «— Это...». В частности, при отображении отношений «— Это...» он предлагает использовать делегирование вместо наследования, если эти отношения описывают роли, которые экземпляры класса могут играть в разные моменты времени. Служащий, не являющийся контролером, может продвинуться по служебной лестнице и перейти с другой должности на должность регистратора рабочего времени или стать контролером системы учета времени, поэтому для описания подобных ролей используем делегирование (рис. 3.6).



Отношения «— Это часть...» («Is-part-of»), указанные на концептуальной диаграмме, представляют собой другой вид структурного отношения, который можно рассматривать на данном этапе проектирования с применением диаграммы классов. Заметим, что отношение «— Это часть...» между объектами `Shift` и `TimekeepingEvent`, как оказалось, отличается от отношений «— Это часть...» между объектами `Shift` и `Break` и между объектами `Break` и `TimekeepingEvent`. Поэтому отложим включение этих отношений в проект до тех пор, пока не выясним отношения внутри всей схемы диаграмм взаимодействия.

Поскольку диаграммы взаимодействия создаются на основе ситуаций использования, воссоздадим следующую реальную ситуацию использования, описывающую типичную работу сотрудника, пользующегося системой учета рабочего времени в течение одного дня.

Ситуация использования:	Сотрудник использует терминал учета времени для отслеживания рабочего времени.
Участник:	Сотрудник.
Цель:	Ввод в систему учета времени информации об уходе и приходе сотрудника.
Синопсис:	Сотрудник собирается начать рабочую смену, уйти на перерыв, вернуться с перерыва или закончить работу. Он идентифицирует себя в системе учета времени и дает ей знать, какой из четырех вариантов он собирается осуществить.
Тип:	Основной и реальный.
Перекрестные ссылки:	Требования R1, R1.1, R1.2, R1.3 и R2.2. Ключевая ситуация использования «Сотрудник использует терминал учета времени».

Ход событий

Сотрудник	Система
1. Сотрудник вставляет свою идентификационную карточку в считывающее устройство терминала системы учета времени	2. Терминал учета времени отображает текущее время, что свидетельствует о готовности к работе. При этом выводится сообщение для сотрудника о поиске идентификационного номера, введенного им через устройство для считывания карточек После нахождения информации о служащем терминал проверяет, разрешено ли сотруднику использовать именно этот терминал учета рабочего времени. Затем терминал предлагает служащему ввести информацию: он начинает рабочую смену, уходит на перерыв, приходит с перерыва или заканчивает работу

Сотрудник	Система
3. Сотрудник указывает на терминале, что он либо начинает рабочую смену, либо уходит на перерыв, либо возвращается с перерыва, либо заканчивает работу	4. Терминал учета рабочего времени производит запись, отражающую выбор сотрудника. При этом подтверждается завершение операции учета рабочего времени, о чем свидетельствует отображение текущего времени и готовность к работе со следующим служащим

В данной ситуации использования участвуют классы, которых нет на рис. 3.6. В этой ситуации использования говорится, что терминал учета рабочего времени взаимодействует с пользователем, поэтому необходимо включить в наш проект класс пользовательского интерфейса.

В ситуации использования говорится также о создании постоянной записи событий системы учета рабочего времени. Чтобы управлять этими данными, а также информацией о служащих, которую должен искать терминал учета рабочего времени, необходимо иметь базу данных. И опять мы оставляем за собой право вносить дополнительные уточнения позднее.

Нужно свести к минимуму количество зависимостей между пользовательским интерфейсом и классами, реализующими внутреннюю логику терминала учета рабочего времени. Поддерживая слабую связь между интерфейсом и внутренней логикой, мы повышаем надежность ПО. Достижению этой цели служит шаблон Facade. Он предназначен для поддержания слабой связи между функционально связанным набором классов и соответствующими клиентскими классами, поэтому между набором классов и соответствующими клиентскими классами вводится дополнительный класс. Почти весь или даже весь доступ клиентов к набору классов происходит через этот класс. Кроме того, в дополнительном классе инкапсулирована общая логика, которая нужна для использования этого набора классов.

Вводимый в проект класс-фасад называется `TimekeepingController`. Он отвечает за управление последовательностью событий во время взаимодействия терминала учета времени с пользователем.

На основании предыдущих ситуаций использования можно создать диаграмму взаимодействия (рис. 3.7).

Опишем взаимодействия, показанные на данной диаграмме.

1. Объект `UserInterface` начинает взаимодействие с передачи идентификационного номера служащего методу `doTransaction` объекта `TimekeepingController`.

   1.1. Объект `TimekeepingController` получает информацию о сотруднике, имеющем данный идентификатор, передавая идентификационный номер служащего методу `LookupEmployee` объекта `Database`. Метод `LookupEmployee` возвращает объект `Employee`,

1.2. Объект `TimekeepingController` вызывает метод `getEventType` объекта `UserInterface`, который заставляет пользовательский интерфейс напомнить сотруднику о типе события, которое должно быть зарегистрировано. Этот метод возвращает значение, обозначающее тип регистрируемого события.

1.3. Объект `TimekeepingController` передает тип события, полученный от пользовательского интерфейса, методу `createEvent` класса `TimekeepingEvent`. Метод `createEvent` возвращает объект, в котором инкапсулировано событие.

1.4. Объект `TimekeepingController` передает объект события методу `storeEvent` объекта `Database` для сохранения его в базе данных.

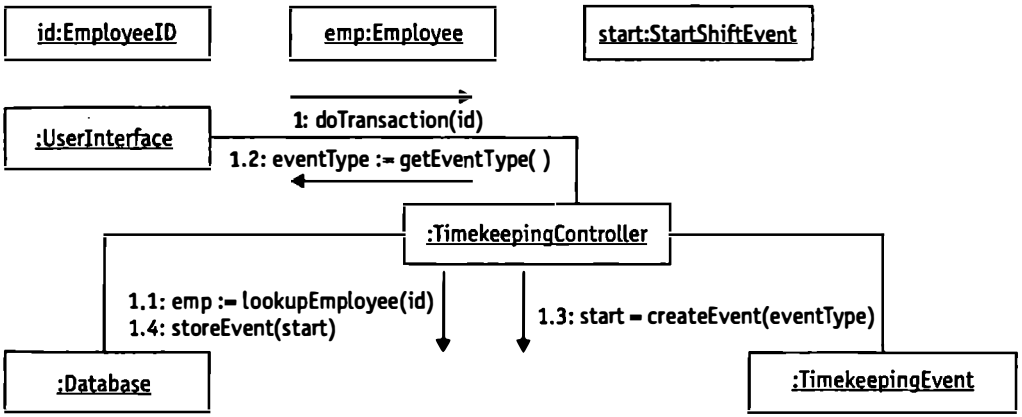


Рис. 3.7. Диаграмма взаимодействия для случая «начало смены»

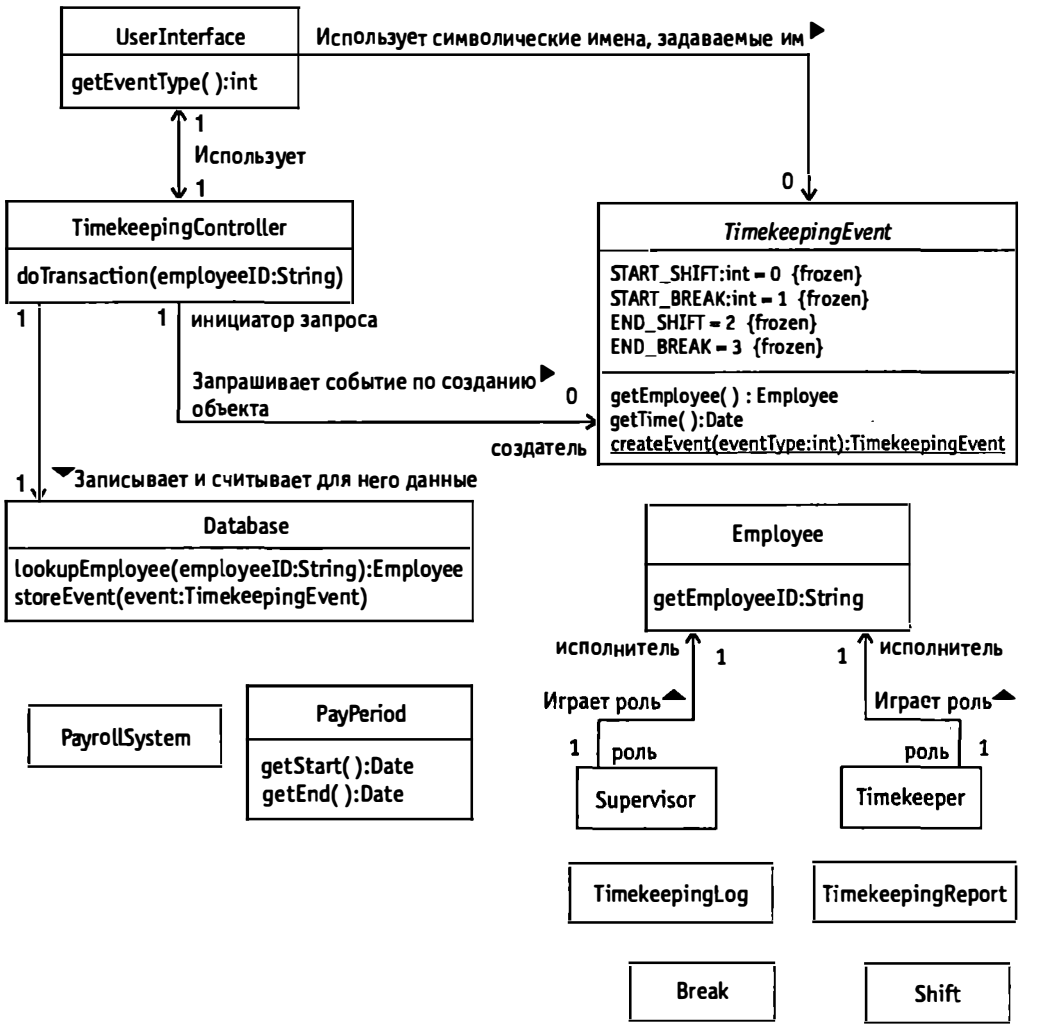
Рассмотрим некоторые особенности структуры диаграммы взаимодействия, представленной на рис. 3.7.

- Операция `lookupEmployee` класса `Database` создает объект для инкапсуляции найденной информации о служащем. Шаблон Creator, описанный в книге [Grand99], утверждает, что если объект содержит, агрегирует, записывает экземпляры класса или осуществляет инициализацию данных для экземпляров класса, то такой объект является хорошим кандидатом на роль создателя экземпляров этого класса. Поэтому объекты, которые создает операция `lookupEmployee` класса `Database`, будут экземплярами класса `Employee`.
- Событие «начало смены» является разновидностью события системы учета рабочего времени, поэтому объекты, представляющие это событие, являются экземплярами подкласса класса `TimekeepingEvent`. Будут существовать также и другие подклассы класса `TimekeepingEvent`, представляющие другие виды событий системы учета рабочего времени. Не нужно, чтобы

класса `TimekeepingEvent`, так как задача — сделать зависимости между пользовательским интерфейсом и внутренней логикой минимальными. Для решения этой задачи используется шаблон Factory Method.

Шаблон Factory Method делает один класс отвечающим за создание экземпляров других классов, имеющих общий суперкласс или реализующих общий интерфейс. В соответствии с этим шаблоном делегируем классу `TimekeepingEvent` обязанность по созданию экземпляров его подклассов.

На рис. 3.8 представлена другая версия нашей диаграммы классов, которую мы уже рассматривали. Новая диаграмма содержит уточнения, о которых мы узнали из диаграммы взаимодействия на рис. 3.6.



Из проекта был удален класс `EmployeeBadge`, поскольку механизм получения идентификационного номера сотрудника — это часть пользовательского интерфейса, а не внутренней логики.

Чтобы внести в проект следующее уточнение, подробнее рассмотрим класс `Database`. Задачей объекта `TimekeepingLog` концептуальной модели являлась поддержка журнала всех событий системы учета рабочего времени. Мы создали соответствующий класс в первоначальном проекте. Отметим, что эта ответственность была возложена на класс `Database`. А это значит, что в нашем проекте класс `TimekeepingLog` не нужен, поэтому удаляем его.

Существует еще одна часть проекта, которую мы не рассматривали, — это создание отчетов по учету рабочего времени. Контролеры используют такие отчеты для просмотра рабочего времени их подчиненных. Специальным образом отформатированные отчеты поступают в систему платежных ведомостей.

В отчетах события системы учета времени объединены в рабочие смены. Смена — промежуток времени, в течение которого служащий находится на работе. Он должен работать всю смену за исключением перерывов — таких промежутков времени, когда служащий прекращает работу, чтобы, например, пообедать, отдохнуть или выкурить сигарету.

При расчете зарплаты служащего время смены делится на три категории:

1. *Обычное время*, оплачиваемое по стандартной почасовой ставке работника.
2. *Сверхнормативное время*, превышающее период обычной занятости служащего. Оно оплачивается по ставке, в несколько раз превышающей ставку оплаты обычного времени.
3. *Неоплачиваемое время*, за которое служащему не платят денег. Некоторые или все перерывы служащего могут быть неоплачиваемым временем.

Отчеты о регистрации рабочего времени должны подразделять рабочее время служащего на эти категории и указывать сумму, заработанную им, без налогов и других вычетов.

Правила классификации при определении, является ли время обычным, сверхнормативным или неоплачиваемым, различны для разных ситуаций, как и правила вычисления множителя, на который умножается обычная ставка при оплате сверхнормативного времени. В настоящее время «Продовольственная сеть Генри» функционирует в пределах только одного штата США. Но разрабатываются планы, которые позволят ей расшириться и развернуть свою деятельность в других штатах. Поэтому проект должен учитывать различные правила, применяемые при классификации рабочего времени и вычислении оплаты труда служащего.

Чтобы иметь возможность выбора различных наборов правил для выполнения вычислений по учету рабочего времени, необходимо определить способ организации набора правил как набора объектов. Набор правил можно представить в виде конечных автоматов, на входе которых — события системы учета рабочего

времени, а откликом на эти входные данные являются вычисления по учету времени. На диаграмме состояний, представленной на рис. 3.9, показан пример такого конечного автомата, который моделирует правила учета рабочего времени для штата Джорджия.

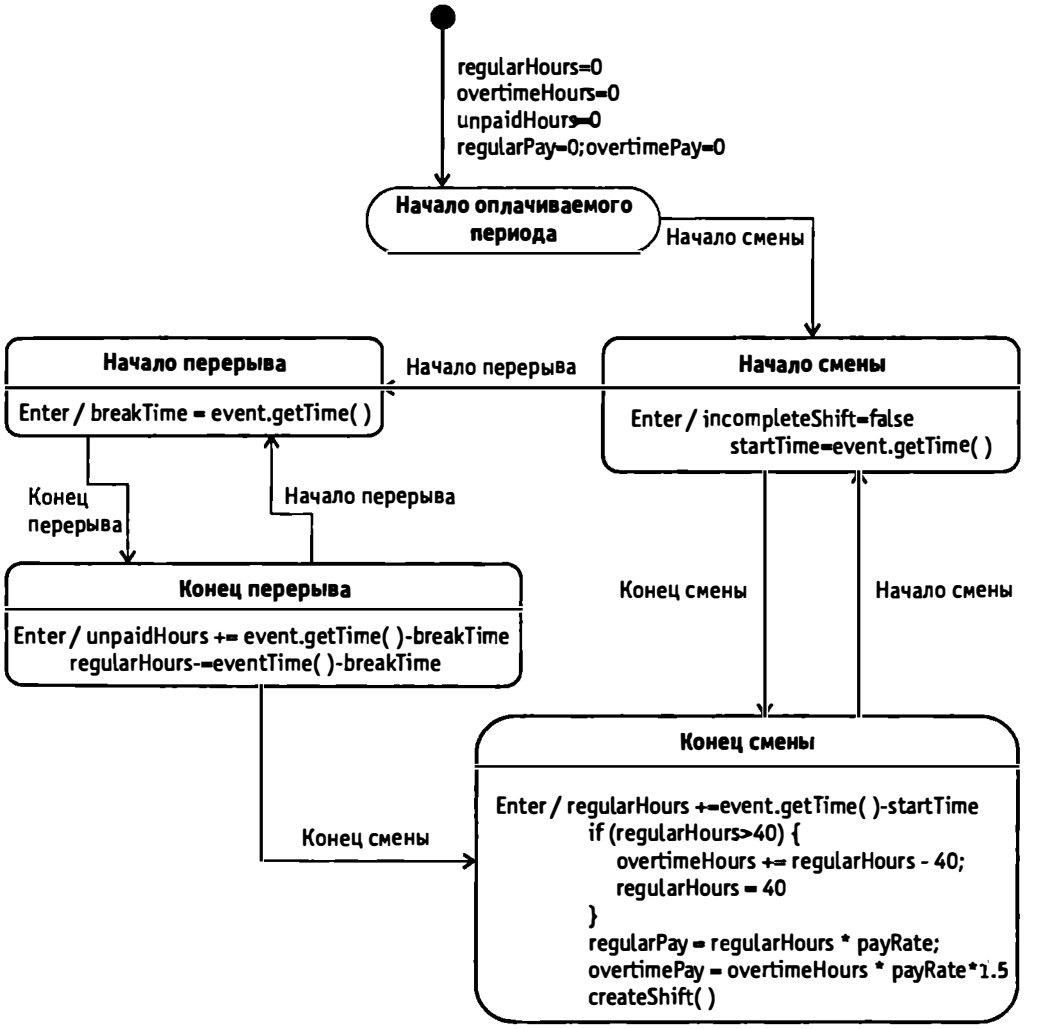


Рис. 3.9. Пример вычислений в системе учета рабочего времени, выполняемых на основе состояний

Для реализации конечного автомата (рис. 3.9) можно воспользоваться шаблоном State, который реализует состояния класса состояний в виде классов, реализующих общий интерфейс. Диаграмма классов, изображенная на рис. 3.10, показывает использование шаблона State для реализации конечного автомата,

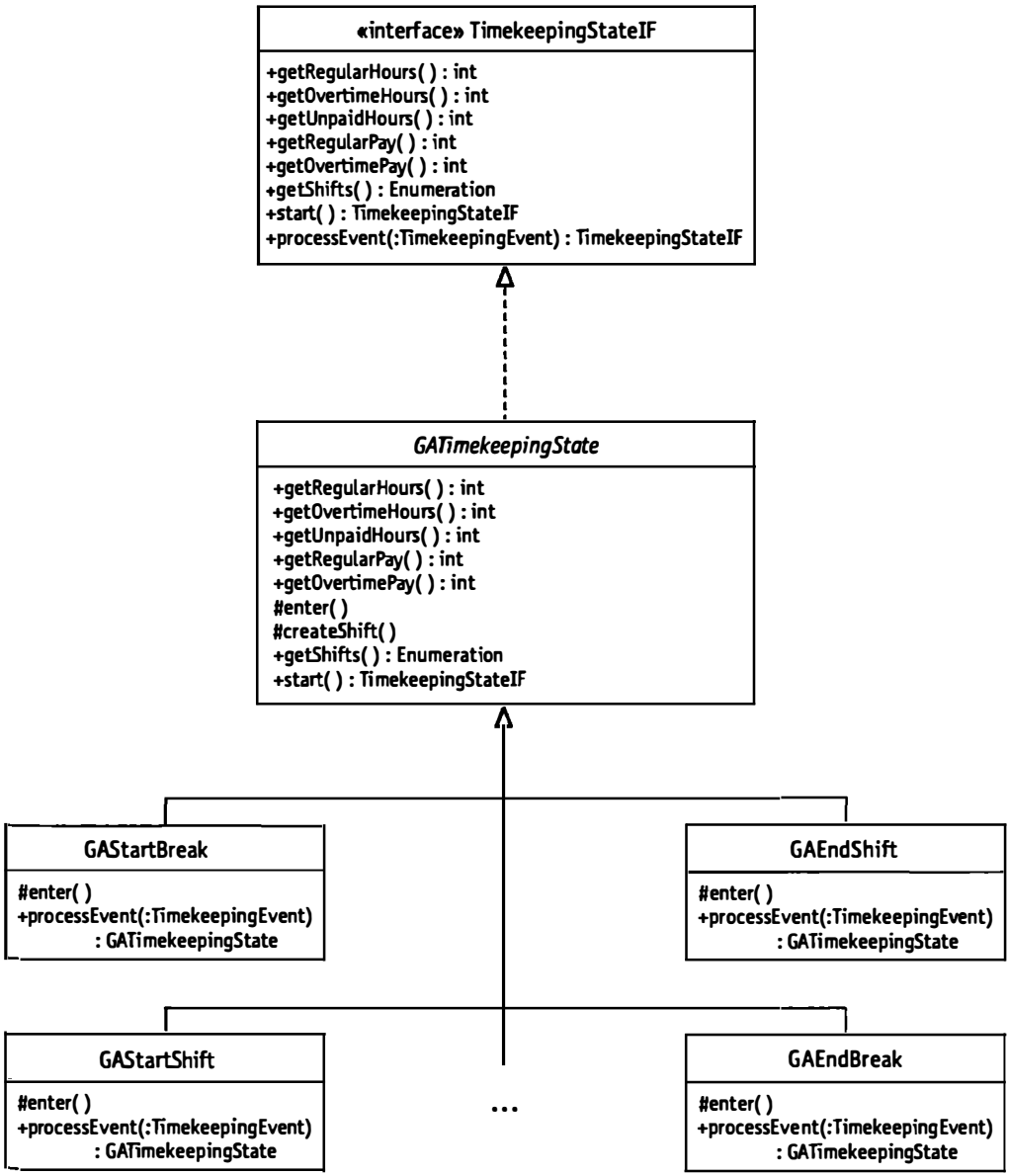


Рис. 3.10. Классы состояний системы учета рабочего времени

Покажем, как используются классы, изображенные на рис. 3.10. Программа, производящая выборку событий системы учета рабочего времени служащих из базы данных, создает объект `GATimekeepingState`. Для представления каждого состояния конечного автомата этот объект использует экземпляр каждого своего подкласса. Метод `start` объекта `GATimekeepingState` возвращает начальное состояние конечного автомата и делает это состояние текущим.

После того как программа создала экземпляр класса `GATimekeepingState`, она вызывает метод `start` этого объекта для считывания начального состояния. Затем программа начинает считывать события системы учета рабочего времени из базы данных. Каждый обнаруженный объект события системы учета рабочего времени программа передает методу `processEvent` объекта текущего состояния. Этот метод заставляет конечный автомат выполнить переход в другое состояние, которое соответствует переданному ему типу события системы учета рабочего времени. Метод `processEvent` возвращает новое текущее состояние. Когда этот метод хочет заставить конечный автомат перейти в некоторое состояние, он вызывает метод `enter` объекта состояний.

По мере перехода конечного автомата из одного состояния в другое он подсчитывает рабочее время служащих и размер зарплаты. Кроме того, он объединяет события системы учета времени, образуя смены. Когда программа заканчивает передачу событий конечному автомату, она может вызвать соответствующие методы объекта `GATimekeepingState` для получения сведений о рабочих часах служащего, размере оплаты или перечне рабочих смен (`Enumeration`).

На этом закончим исследование данной проблемы. Теперь вы должны иметь представление об использовании шаблонов проектирования.

ГЛАВА

Основные шаблоны проектирования

Шаблоны, представленные в данной главе, самые фундаментальные и важные. Они активно используются другими шаблонами.

Шаблоны Delegation, Interface, Abstract Superclass и Interface and Abstract Class показывают, как организовать отношения между классами. Большинство шаблонов используют хотя бы один из этих шаблонов. Они настолько популярны, что часто даже не упоминаются в разделе «Шаблоны, связанные с данным шаблоном проектирования» при описании большинства шаблонов.

Шаблон Immutable описывает, как избежать ошибок и задержек во время доступа многочисленных объектов к одному и тому же объекту. Хотя шаблон Immutable явно не является частью подавляющего большинства других шаблонов, он используется преимущественно с другими шаблонами.

Шаблон Marker Interface рассматривает способ упрощения проектирования классов, которые имеют постоянный атрибут типа boolean.

Шаблон Proxy является основой для многих шаблонов, реализующих логику управления доступом одного объекта к другому достаточно «прозрачным» способом.

Delegation (Делегирование)

(Когда не используется наследование)

СИНОПСИС

В некоторых случаях использование наследования приводит к созданию неудачного проекта. Хотя и менее удобное, делегирование представляет собой универсальный способ расширения классов. Делегирование применимо во многих ситуациях, где наследование терпит фиаско.

КОНТЕКСТ

Наследование — распространенный способ расширения и многократного использования функциональности класса. Делегирование представляет собой более общий подход к решению задачи расширения возможностей поведения класса. Этот подход заключается в том, что некоторый класс вызывает методы другого класса, а не наследует их. Во многих ситуациях, не позволяющих использовать наследование, возможно применение делегирования.

Наследование, например, подходит для описания отношений «is-a-kind-of» («Это разновидность...»), так как они очень статичны по своей природе. Однако отношения «is-a-role-played-by» («Это роль, которую играет...») неудобно моделировать при помощи наследования. Экземпляры класса могут играть сразу несколько ролей. Рассмотрим в качестве примера систему заказа авиабилетов. Для нее характерны такие роли: пассажир, агент по продаже билетов и персонал, обслуживающий рейс. Эти роли можно представить следующим образом: класс с именем `Person` имеет подклассы, соответствующие данным ролям (рис. 4.1).

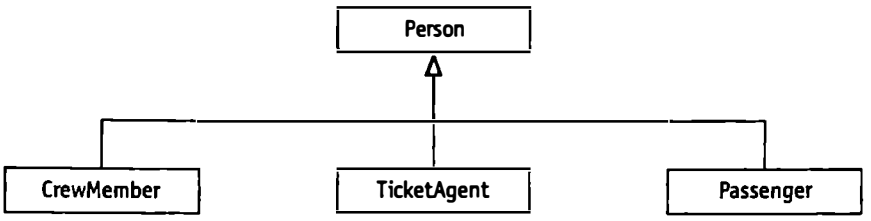


Рис. 4.1. Моделирование ролей при помощи наследования

Проблема, связанная с изображенной на рис. 4.1 диаграммой, состоит в том, что один и тот же субъект может играть несколько ролей. Субъект, который

входит в обслуживающий персонал, может быть также и пассажиром. Некоторые авиалинии иногда направляют персонал компании на работу по учету билетов. Это значит, что один и тот же субъект может играть любую из указанных ролей. При моделировании этой ситуации для класса `Person` необходимо создать семь подклассов (рис. 4.2). Количество нужных подклассов экспоненциально возрастает при увеличении количества ролей. Например, чтобы смоделировать все возможные комбинации для шести ролей, необходимо создать 63 подкласса.

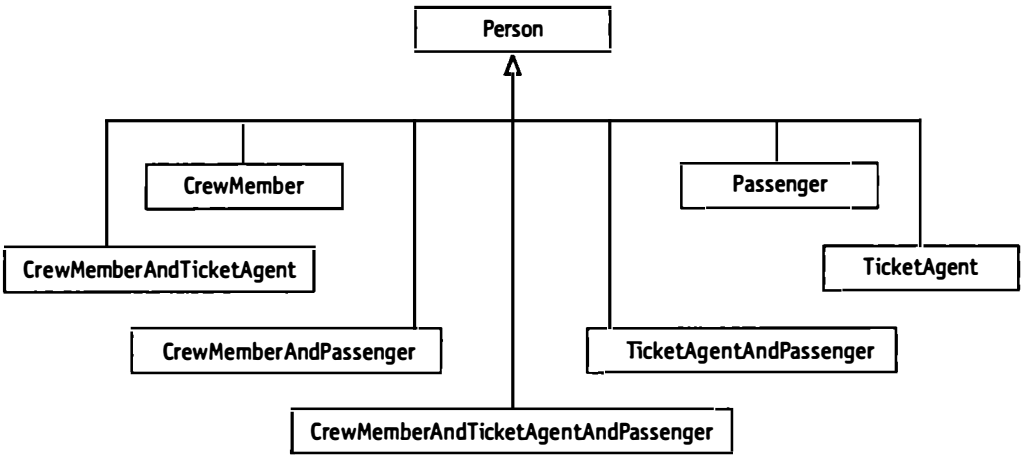


Рис. 4.2. Моделирование множества ролей при помощи наследования

Более серьезная проблема заключается в том, что один и тот же субъект может играть различные комбинации ролей в разное время. Отношения, описываемые наследованием, являются статичными и не меняются со временем. Чтобы в условиях использования наследования смоделировать различные комбинации ролей на протяжении некоторого времени, один и тот же субъект должен быть представлен множеством разных объектов в разные моменты времени. Моделирование динамически изменяющихся ролей при помощи наследования вызывает большие затруднения.

С другой стороны, представление одного и того же субъекта, играющего различные роли, можно осуществить при помощи делегирования, причем все проблемы решаются сами собой. Рис. 4.3 показывает, как нужно реорганизовать модель, используя делегирование.

Согласно данной структуре, объект `Person` делегирует ответственность по выполнению определенной роли некоторому объекту, который специально предназначен для выполнения этой роли. Таким образом, в системе вам потребуется определить столько объектов, сколько существует ролей. Различные их комбинации не потребуют дополнительных объектов. Поскольку объекты делегирования могут быть динамичными, объекты ролей могут добавляться или удаляться в соответствии с исполнением субъектом различных ролей.

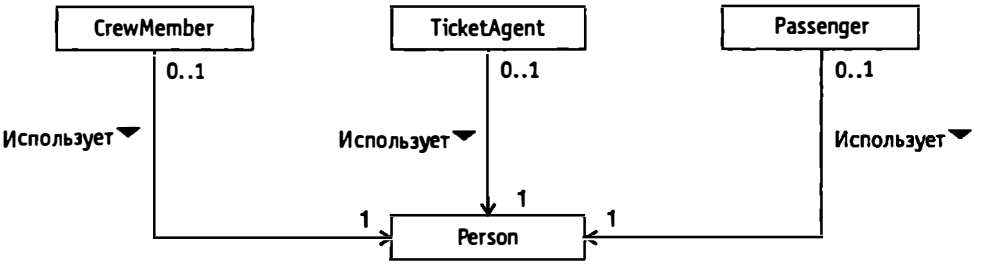


Рис. 4.3. Моделирование ролей при помощи делегирования

В случае с системой резервирования авиабилетов заранее определенный набор ролевых объектов может связываться с различными объектами `Person` через какое-то время. Например, при планировании рейса было определено, что на борту должно быть некоторое количество членов экипажа. При подготовке рейса определенные субъекты могут быть связаны с ролями членов экипажа. Когда расписание меняется, субъект может быть переведен с роли члена экипажа на другую.

МОТИВЫ

☺ Наследование — это статическое отношение, которое не меняется со временем. Если оказалось, что в разные моменты времени объект должен быть представлен разными подклассами одного и того же класса, то данный объект нельзя представить подклассом этого общего класса. Если объект создается как экземпляр какого-то класса, то он всегда будет экземпляром этого класса. С другой стороны, в разные моменты времени объект может делегировать полномочия по своему поведению другим объектам.

☺ Если класс пытается сокрыть от других классов метод или переменную, унаследованную им от суперкласса, то этот класс не должен наследоваться от такого суперкласса. Не существует способа эффективного сокрытия методов и переменных, унаследованных от суперкласса. С другой стороны, имеется возможность использования объектом методов и переменных другого объекта при условии, что он является единственным объектом, который имеет доступ к другому объекту. Это очень похоже на наследование, но здесь используются динамические отношения, которые могут меняться со временем.

☺ «Функциональный» класс (класс, имеющий отношение к функциональности программы) не должен быть подклассом вспомогательного класса. Существуют по крайней мере две причины не делать этого:

1. При объявлении класса подклассом таких классов, как `ArrayList` или `HashMap`, есть риск того, что эти классы могут измениться впоследствии и стать несовместимыми с предыдущими версиями. Хотя риск этого невелик, обычно нет таких уж видимых причин идти на этот риск.

2. Когда «функциональный» класс создается как подкласс вспомогательного класса, то, как правило, необходимо использовать функциональность вспомогательного класса для реализации функциональности, присущей самой задаче. Делая это с помощью наследования, мы ослабляем инкапсуляцию «функционального» класса.

Клиентские классы будут использовать наш «функциональный» класс, исходя из предположения, что он наследуется каким-то определенным вспомогательным классом. Пусть мы решили поменять реализацию нашего «функционального» класса, наследуя его от другого вспомогательного класса. Тогда ранее написанные клиентские классы станут непригодными.

Еще одна, даже более серьезная проблема состоит в том, что клиентские классы могут вызывать открытые методы вспомогательного суперкласса, нарушая при этом инкапсуляцию нашего класса, унаследованного от этого суперкласса.

☹ Делегирование может быть менее удобным, чем наследование, ввиду того что при реализации придется написать больше кода.

☹ Делегирование делает класс менее структурированным, чем наследование. В тех проектах, где структура классов важна, эта структурированность и отсутствие гибкости у наследования могут быть достоинством. Это часто справедливо для каркасов (frameworks) приложений. Более подробно этот вопрос обсуждается при рассмотрении шаблона Template Method (см. главу 8).

Некоторые случаи неправильного использования наследования настолько популярны, что их можно рассматривать как антишаблоны. В частности, наследование от вспомогательных классов и использование наследования для моделирования ролей представляют собой широко распространенные ошибки проектирования.

☺ Для многократного использования класса или его расширения чаще используется делегирование.

☺ Поведение, которое класс наследует от своего суперкласса, не может быть с легкостью изменено. Наследование не может быть использовано, если поведение класса не может быть четко определено на этапе проектирования.

РЕШЕНИЕ

Применяйте делегирование для многократного использования характеристик поведения и расширения класса. Делегирование реализуется посредством написания нового класса (делегата), который включает в себя функциональность исходного класса (делегируемого), используя его экземпляр и вызывая его методы (рис. 4.4).

Рис. 4.4. Делегирование

Данная диаграмма показывает, что класс, выступающий в роли `Delegator`, использует класс, выступающий в роли `Delegatee`.

Делегирование решает более общие задачи, чем наследование. Любое расширение класса, которое может быть выполнено при помощи наследования, может также быть выполнено при помощи делегирования.

РЕАЛИЗАЦИЯ

Реализовать делегирование несложно. Для этой цели просто используется ссылка на экземпляр класса.

Самый лучший способ убедиться, что делегирование будет легко реализовать, — это сделать явными его структуру и назначение. Одним из решений этой задачи является реализация делегирования с использованием шаблона Interface.

СЛЕДСТВИЯ

При использовании делегирования не возникает проблем, сопутствующих наследованию. Преимущество делегирования заключается в простоте определения поведения на стадии выполнения.

Основным недостатком делегирования является его меньшая структурированность, чем наследования. Отношения между классами, построенные с применением делегирования, менее очевидны, чем представленные при помощи наследования. Для более наглядного отображения основанных на делегировании отношений применяют следующие стратегии:

- При ссылке на объекты, играющие определенную роль, используют схемы со смысловыми названиями. Например, если несколько классов делегируют создание элементов управления окном (widget), то роль объекта делегирования становится более очевидной, если все классы, делегирующие эту операцию, ссылаются на объекты делегирования через переменную `widgetFactory`.
- Операцию делегирования всегда можно снабдить комментарием.
- Шаблон Law of Demeter (описанный в книге [Grand99]) утверждает, что если без делегирования класс имеет только косвенную связь с другим классом, то делегирование должно быть косвенным. Не делегируют прямым образом поведение классу, связанному косвенно и предоставляющему характеристики поведения. Вместо этого делегируют поведение классу, связанному прямым образом, и убеждаются, что он делегирует поведение именно тому классу, который отвечает за поведение. При этом упрощается весь процесс проектирования, так как количество связей между объектами становится минимальным. Однако в отдельных случаях такое косвенное делегирование может сделать вспомогательный класс менее понятным из-за добавления методов, не связанных с прямым назначением этого класса. В таких случаях

переместите несвязанные данные в отдельный класс, используя шаблон Pure Fabrication (также описанный в книге [Grand99]).
- Используют хорошо известные шаблоны проектирования и кодирования. Человек, читающий текст программы, в которой применяется делегирование, скорее поймет исполняемые объектами роли, если они являются частью хорошо известного или часто встречающегося в программе шаблона.

Заметим, что возможно и желательно использовать все эти стратегии одновременно.

ПРИМЕНЕНИЕ В JAVA API

В Java API существует множество примеров делегирования. Это представляет собой основу модели делегирования событий в языке Java. Согласно этой модел объекты источника событий отправляют события объектам приемника собы тий. Объекты источника событий обычно не принимают решения, что делать с событием. Вместо этого они делегируют обязанность по обработке события объектам приемника.

ПРИМЕР КОДА

В качестве примера делегирования рассмотрим систему, которая отвечает за отслеживание проверенных частей багажа. Допустим, система включает классы представляющие сегмент рейса[1], багажное отделение и части багажа, как показано на рис. 4.5.

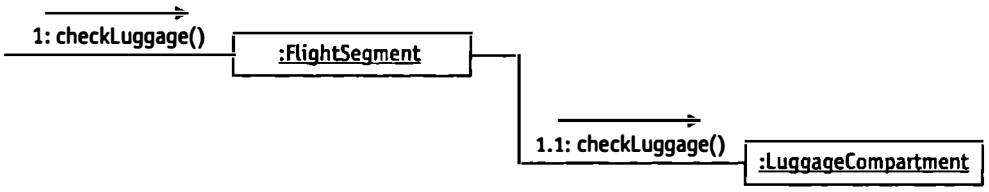


Рис. 4.5. Проверка багажа

Класс `FlightSegment` содержит метод `checkLuggage`, который контролируе часть багажа, оформляемого на рейс (рис. 4.5). Класс рейса делегирует эту опе рацию экземпляру класса `LuggageCompartment`.

Кроме того, делегирование широко применяется при реализации коллекции Рассмотрим диаграмму класса, представленную на рис. 4.6.

Класс `LuggageCompartment` поддерживает коллекцию других объектов. Клас сы проблемной области, которые отвечают за поддержку коллекции други

[1] Сегмент рейса — это часть путешествия без пересадки.

объектов, обычно делегируют эту ответственность другому объекту, например, экземпляру класса `java.util.ArrayList`. Поскольку реализация коллекции при помощи делегирования очень хорошо известна, отдельный класс коллекции обычно не указывается на схемах проектирования.

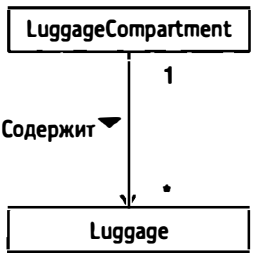


Рис. 4.6. Багажное отделение

Приведем фрагменты кодов, которые реализуют проект, представленный на рис. 4.5. Сначала покажем класс `FlightSegment`, делегирующий операцию `checkLuggage` классу `LuggageCompartment`:

```
class FlightSegment {
...
   LuggageCompartment luggage;
...
   void checkLuggage(Luggage piece) throws LuggageException {
      luggage.checkLuggage(piece);
   } // checkLuggage(Luggage)
} // class FlightSegment
```

А теперь реализуем класс `LuggageCompartment`, делегирующий коллекцию частей багажа классу `ArrayList`:

```
class LuggageCompartment {
...
   // Части багажа в классе LuggageCompartment.
   private ArrayList pieces = new ArrayList();
...
   void checkLuggage(Luggage piece) throws LuggageException {
...
      pieces.add(piece);
   } // checkLuggage(Luggage)
} // class LuggageCompartment
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ DELEGATION

Почти все шаблоны используют делегирование. Некоторые шаблоны почти целиком основываются на делегировании, к ним можно отнести шаблоны Decorator и Proxy.

Кроме того, шаблон Interface может быть полезен для того, чтобы сделать структуру делегирования и мотивацию его применения более понятными и простыми для программистов, занимающихся поддержкой.

Interface (Интерфейс)

СИНОПСИС

Экземпляры некоторого класса предоставляют данные и сервисы экземплярам других классов. Чтобы клиентские классы не зависели от конкретных классов, предоставляющих данные и сервисы, необходимо заменить их другим классом, предоставляющим данные и сервисы, но имеющим минимальное влияние на клиентские классы. Это реализуется следующим образом: другие классы получают доступ к данным и сервисам через некоторый интерфейс.

КОНТЕКСТ

Предположим, создается приложение, которое управляет закупкой товаров для бизнеса. Программа должна содержать информацию о таких объектах, как, например, поставщики товаров, транспортные компании, получающие товар торговые точки, бухгалтерии. У всех этих объектов есть один общий аспект — они имеют адреса с названиями улиц. Эти адреса могут встречаться в различных частях программы. Нужно создать класс, который сможет отображать и редактировать адреса, чтобы можно было использовать его в любом месте программы. Назовем этот класс `AddressPanel`.

Необходимо, чтобы объекты класса `AddressPanel` могли получать и устанавливать информацию об адресе бизнес-объекта в отдельном объекте. Очевидно, что продавцы, транспортные компании и т.д. должны быть представлены разными классами. При этом возникает вопрос, как класс `AddressPanel` будет взаимодействовать с нашими бизнес-объектами. Для решения данной проблемы создадим интерфейс, в котором будут методы для задания информации об адресе участника торговых отношений и методы для получения данной информации.

Тогда для того, чтобы экземпляры класса `AddressPanel` могли взаимодействовать с бизнес-объектами системы, необходимо, чтобы эти бизнес-объекты имплементировали наш интерфейс.

Используя предоставляемую интерфейсом косвенность, клиенты класса `AddressPanel` могут обращаться к методам объекта данных, не зная, к какому конкретно классу он принадлежит. На рис. 4.7 изображена диаграмма классов, на которой представлены такие отношения.

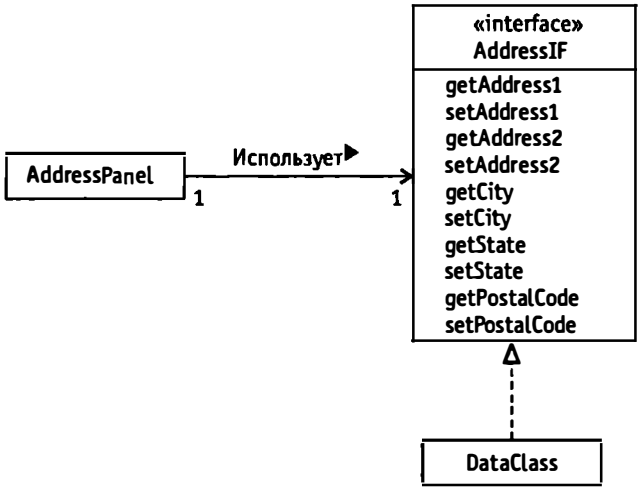


Рис. 4.7. Косвенность, осуществляемая через адресный интерфейс

МОТИВЫ

☺ Некоторый объект использует другой объект для получения от него данны или сервисов. Если наш объект для получения доступа к другому объект должен явно указать, к какому классу этот объект принадлежит, то возмож ность многократного использования нашего класса ухудшается из-за силь ной связанности.

☺ Нужно изменить объект, используемый другими объектами, и при этом нельзя, чтобы эти изменения затронули какой-либо класс, кроме класса из меняемого объекта.

☹ Конструкторы класса не могут быть доступны через интерфейс, потом что интерфейсы в Java не имеют конструкторов.

РЕШЕНИЕ

Чтобы избежать зависимости классов (когда один использует другой), исполь зуют интерфейсы, делая такую зависимость косвенной (рис. 4.8).

Опишем роли, исполняемые классами и интерфейсами.

Client. Класс `Client` использует другие классы, которые реализуют интерфей `IndirectionIF`.

IndirectionIF. Интерфейс `IndirectionIF` обеспечивает косвенность, что по зволяет классу `Client` быть независимым от реализации класса, играющег роль `Service`. Как правило, интерфейсы, выступающие в такой роли, являютс открытыми.

Service. Класс в этой роли обслуживает классы, выступающие в роли `Client`

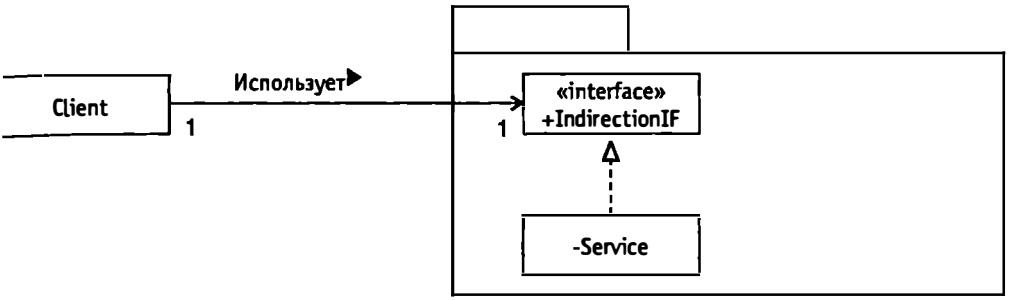


Рис. 4.8. Шаблон Interface

Классы, играющие эту роль, в идеале являются закрытыми для классов вне свого пакета. Если классы `Service` объявляются как закрытые, то для получения доступа к ним классы, не относящиеся к данному пакету, вынуждены использовать интерфейсы.

РЕАЛИЗАЦИЯ

Реализация шаблона Interface проста: определяем интерфейс для предоставления сервиса, пишем клиентские классы для доступа к сервису через интерфейс и создаем классы, предоставляющие сервис и реализующие интерфейс.

У Java-интерфейсов нет конструкторов, поэтому эти интерфейсы нельзя использовать для сохранения ответственности класса за создание независимых от него объектов. В Java API есть класс `java.lang.reflect.Constructor`, который можно использовать для создания объектов, не зная, экземплярами какого класса они будут.

СЛЕДСТВИЯ

☺ При использовании шаблона Interface класс, нуждающийся в сервисе со стороны другого класса, не является больше привязанным к какой-либо конкретной реализации другого класса.

☹ Подобно любой другой косвенности, шаблон Interface может усложнить программу для понимания.

ПРИМЕНЕНИЕ В JAVA API

Java API определяет интерфейс `java.io.FilenameFilter`. Этот интерфейс объявляет метод `accept`, который принимает в качестве аргумента имя файла. Этот метод возвращает true или false в зависимости от того, должен ли быть включен в список указанный файл. Java API определяет также класс `java.awt.FileDialog`, который может использовать объект `FilenameFilter` для фильтрации файлов для их последующего использования.

ПРИМЕР КОДА

Примером шаблона Interface является класс `AddressPanel`. Интерфейс `AddressIF` рассматривается в разделе «Контекст». Приведем код для класса `AddressPanel`:

```
class AddressPanel extends Panel (
   private AddressIF data;      // Объект данных.
...
   /**
    * Зададим объект данных, с которым будет работать эта панель.
    */
   public void setData(AddressIF address) {
...
   } // setData(AddressIF)

   /**
    * Сохраним содержимое TextFields в объекте данных.
    */
   public void save() {
      if (data != null) {
         data.setAddress1(address1Field.getText());
...
         data.setPostalCode(postalCodeField.getText());
      } // if data
   } // save()
} // class AddressPanel
```

Заметьте, что при объявлении переменной `data` `AddressPanel` класса мы указали в качестве типа переменной не класс, а интерфейс.

Назначение шаблона Interface заключается в предоставлении интерфейса, обеспечивающего косвенность между клиентским и сервисным классами. Приведем код для интерфейса `AddressIF`:

```
public interface AddressIF {
   public String getAddress1();
   public void setAddress1(String address1);
...
   public String getPostalCode();
   public void setPostalCode(String PostalCode);
} // interface AddressIF
```

Интерфейс просто объявляет методы, которые требуются для получения необходимой информации.

Ниже приведен код сервисного класса. Влияние шаблона Interface на класс ограничивается тем, что класс должен реализовывать интерфейс `AddressIF`.

```
class ReceivingLocation extends Facility implements AddressIF{
   private String address1;
...
   private String postalCode;
...
   public String getAddress1() { return address1; }
   public void setAddress1(String address1) {
      this.address1 = address1;
   } // setAddress1(String)
...
   public String getPostalCode() { return postalCode; }
   public void setPostalCode(String postalCode) {
      this.postalCode = postalCode;
   } // setPostalCode(String)
} // class ReceivingLocation
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ INTERFACE

Delegation. Шаблоны Delegation и Interface часто используются вместе.

Adapter. Шаблон Adapter позволяет объектам, ожидающим от другого объекта, что он реализует определенный интерфейс, работать с объектами, не реализующими предполагаемый интерфейс.

Strategy. Шаблон Strategy использует шаблон Interface.

Anonymous Adapter. Шаблон Anonymous Adapter (описанный в книге [Grand99]) использует шаблон Interface.

Abstract Superclass (Абстрактный суперкласс)

Этот шаблон впервые был описан в работе [Rhiel2000].

СИНОПСИС

Гарантирует согласованное поведение концептуально связанных классов, задавая для них общий абстрактный суперкласс.

КОНТЕКСТ

Нужно написать классы для предоставления последовательного доступа (только для чтения) к некоторым структурам данных. Допустим, эти классы будут реализовывать интерфейс `java.util.Iterator`.

Интерфейс `Iterator` содержит метод, который называется `remove`. В документации сказано, что назначение метода `remove` — удалять объекты из источника после их извлечения. В описании метода `remove` сказано также, что этот метод необязательный. Реализация данного метода может просто генерировать исключение `UnsupportedOperationException`.

Цель заключается в том, чтобы предоставить доступ только для чтения к структурам данных, поэтому нужно, чтобы все классы генерировали исключение `UnsupportedOperationException`. При вызове метода `remove`, чтобы быть уверенным, что все классы реализуют метод `remove` совершенно одинаково, необходимо создать общий абстрактный класс для всех классов `Iterator`, от которого они будут унаследованы. Общий суперкласс реализует метод `remove`, который генерирует исключение `UnsupportedOperationException` при вызове этого метода (рис. 4.9).

МОТИВЫ

☺ Нужно гарантировать, чтобы общая логика для связанных классов реализовывалась одинаково для каждого класса.

☺ Нужно избежать издержек, связанных со временем выполнения и поддержкой излишнего кода.

☺ Нужно упростить написание связанных классов.

☹ Нужно задать общее поведение, хотя во многих ситуациях наследование не самый подходящий способ его реализации (см. шаблон Delegation).

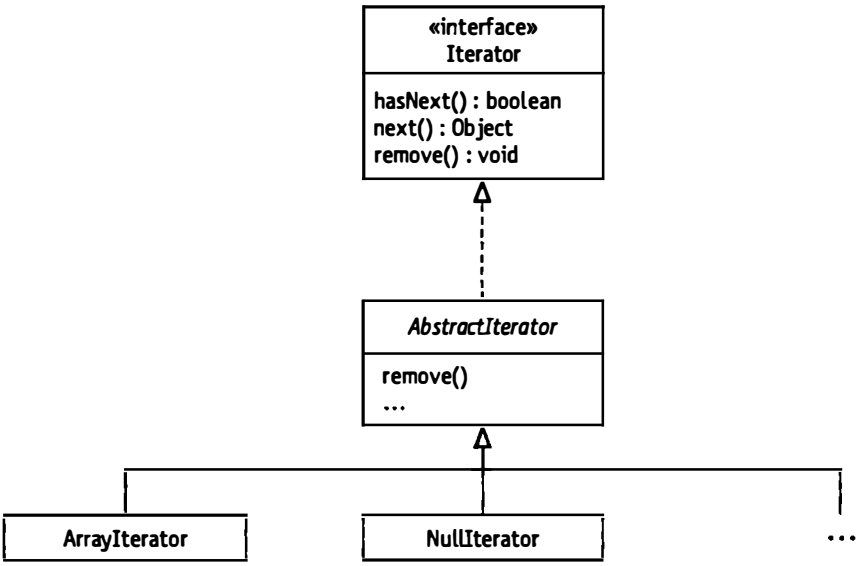


Рис. 4.9. Итераторы и их абстрактный суперкласс

РЕШЕНИЕ

Реализуем общую логику связанных классов в суперклассе. Варианты поведения, зависящие от конкретного наследника, поместим в методы с одинаковой сигнатурой. Сделаем эти методы абстрактными в нашем суперклассе. На рис. 4.10 представлена подобная структура.

Опишем роли, которые играют классы в рамках шаблона Abstract Superclass.

Abstract Superclass. Класс, выступающий в этой роли, представляет собой абстрактный суперкласс, в котором инкапсулирована общая логика связанных классов. Связанные классы расширяют этот класс. Таким образом, они могут наследовать его методы. Методы с одинаковыми сигнатурами и общей логикой для всех связанных классов помещаются в суперкласс, поэтому логика этих методов может наследоваться всеми подклассами данного суперкласса. Методы с зависящей от конкретного подкласса данного суперкласса логикой, но с одинаковыми сигнатурами, объявляются в абстрактном классе как абстрактные методы, тем самым гарантируя, что каждый конкретный подкласс будет иметь методы с такими же сигнатурами.

ConcreteClass1, ConcreteClass2 и т.д. Класс, выступающий в этой роли, представляет собой конкретный класс, чья логика и назначение связаны с другими конкретными классами. Методы, общие для этих связанных классов, помещаются в абстрактный суперкласс.

Общая логика, которая не представлена в общих методах, помещается в общие

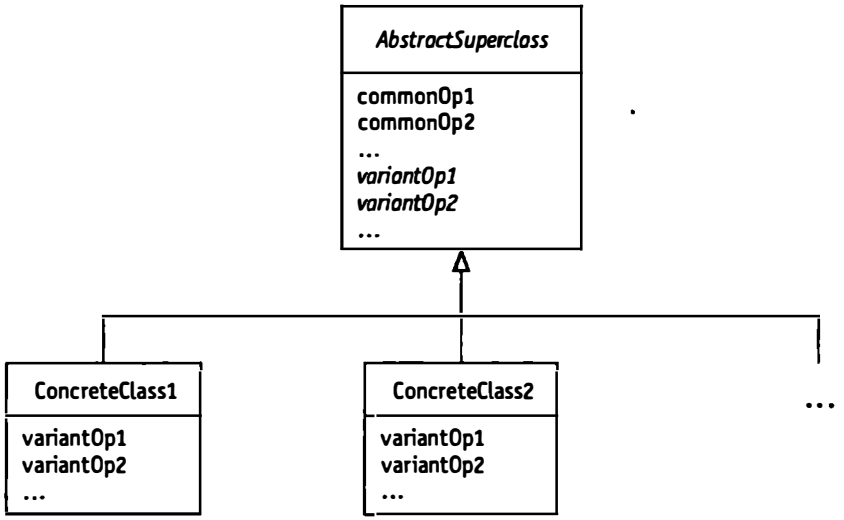


Рис. 4.10. Шаблон Abstract Superclass

РЕАЛИЗАЦИЯ

Если общие методы являются открытыми, то можно поместить эти методы в интерфейс, а абстрактный суперкласс будет имплементировать этот интерфейс.

СЛЕДСТВИЯ

☺ Тестирование будет занимать меньше времени, так как уменьшается количество кода.

☹ Использование шаблона Abstract Superclass приводит к появлению зависимости между суперклассом и его подклассами. Изменение суперкласса может иметь нежелательные последствия для некоторых подклассов.

ПРИМЕНЕНИЕ В JAVA API

Класс `java.awt.AWTEvent` — это абстрактный класс для классов, инкапсулирующих события, связанные с GUI (Graphical User Interface, графический интерфейс пользователя). Он определяет несколько методов, которые являются общими для классов событий.

ПРИМЕР КОДА

В качестве примера рассмотрим реализацию классов из раздела «Контекст». Реализация этих классов взята из ПО фирмы ClickBlocks, которое вы можете найти на сайте этой книги в пакете `org.clickblocks.util`.

Листинг класса `AbstractIterator`:

```
/**
 * Этот абстрактный класс очень удобен,
 * поскольку позволяет определять итераторы
 * путем реализации только одного метода getNextElement().
 */
abstract public class AbstractIterator implements Iterator {
  private Object nextElement;

  /**
   * Этот метод должен вызываться конструктором подкласса.
   */
  protected void init() {
    nextElement = getNextElement();
  } // init()

  /**
   * Этот метод возвращает следующий элемент
   * в просматриваемой структуре данных.
   * Если следующего элемента нет, то возвращает сам себя.
   */
  public abstract Object getNextElement();

  /**
   * Возвращает true, если в структуре данной итерации
   * есть еще не пройденные элементы.
   */
  public boolean hasNext(){
    return nextElement!=this;
  } // hasNext()

  /**
   * Возвращает следующий элемент в итерации.
   *
   * @exception NoSuchElementException Итерация не имеет больше
   * элементов.
   */

  public Object next(){
    if (nextElement==this) {
```

```
            throw new NoSuchElementException();
        } // if
        Object previous = nextElement;
        nextElement = getNextElement();
        return previous;
    } // next()

    /**
     * Удаляет из базовой коллекции последний элемент, возвращенный
     * методом next.
     *
     * @exception UnsupportedOperationException
     * Если операция удаления не поддерживается этим
     * итератором.
     */
    public void remove(){
        throw new UnsupportedOperationException();
    } // remove()
} // class AbstractIterator
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ ABSTRACT SUPERCLASS

Interface and Abstract Class. Шаблон Interface and Abstract Class использует шаблон Abstract Superclass.

Template Method. Шаблон Template Method использует шаблон Abstract Superclass.

Interface and Abstract Class (Интерфейс и абстрактный класс)

СИНОПСИС

Необходимо сделать клиентские классы независимыми от тех классов, которые реализуют поведение, и обеспечить согласованность поведения между классами, реализующими это поведение. Не нужно делать выбор между использованием интерфейса и абстрактного класса. Можно иметь классы, реализующие некоторый интерфейс и наследующиеся от абстрактного класса.

КОНТЕКСТ

Предположим, при проектировании приложения нужно сокрыть класс или классы, которые реализуют некоторое поведение, и поэтому их делают закрытыми и реализующими некий открытый интерфейс. Кроме того, чтобы такая реализация была логичной и удобной, классы должны наследоваться от общего абстрактного класса. Но неизвестно, что сделать предком этих классов — интерфейс или абстрактный класс.

МОТИВЫ

☺ При помощи шаблона Interface интерфейсы в языке Java могут использоваться для сокрытия конкретного класса, реализующего поведение, от клиентов этого класса.

☺ Инкапсуляция общей логики с помощью шаблона Abstract Superclass в суперклассе при написании классов помогает обеспечить связанность реализации. Можно также снизить объем работ по имплементации с точки зрения повторного использования кода.

☹ Если специалисты имеют возможность использовать два разных способа улучшения организации классов, то, как правило, тенденция такова, что нужно выбирать либо тот, либо другой.

РЕШЕНИЕ

Шаблон Interface используют в том случае, если нужно сокрыть от клиентов класс некоторого объекта, предоставляющего определенный сервис. Определяют косвенный доступ клиентских объектов к объекту, предоставляющему сервис, т.е. через интерфейс. Косвенность позволяет клиентам иметь доступ к объекту, предоставляющему сервис, ничего не зная о том, с объектами какого рода они имеют дело.

Если нужно проектировать набор связанных классов, обеспечивающих похожую функциональность, то определяют общие части их реализации в абстрактном суперклассе.

При возникновении двух этих требований в рамках одной задачи используют и интерфейс, и абстрактный класс (рис. 4.11).

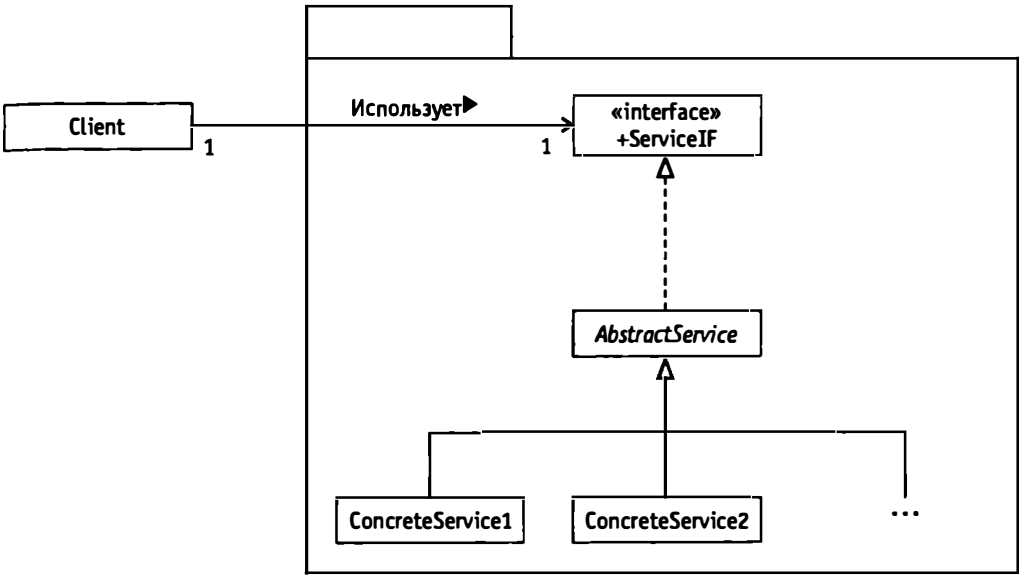


Рис. 4.11. Интерфейс и абстрактный класс

Если применяется подобная комбинация интерфейса и абстрактного класса, интерфейс должен быть открытым, а абстрактный класс, если существует такая возможность, — закрытым.

СЛЕДСТВИЕ

☺ Использование шаблона Interface and Abstract Class позволяет получить все преимущества интерфейсов и абстрактных классов.

ПРИМЕНЕНИЕ В JAVA API

Пакет `javax.swing.table` содержит интерфейсы и классы для работы с таблицами при проектировании интерфейса пользователя. Для каждой таблицы соответствующий объект модели данных содержит значения, отображаемые в таблице. Чтобы быть использованным в качестве модели данных для таблицы, объект должен быть экземпляром некоторого класса, реализующего интерфейс

javax.swing.table.TableModel. Кроме того, в этом пакете содержится класс AbstractTableModel. AbstractTableModel — это абстрактный класс, который содержит некоторую логику, задаваемую по умолчанию и полезную при реализации методов, объявленных в интерфейсе TableModel. И наконец, существует конкретный класс DefaultTableModel, который инстанциируется по умолчанию при создании таблицы. Эти отношения показаны на рис. 4.12.

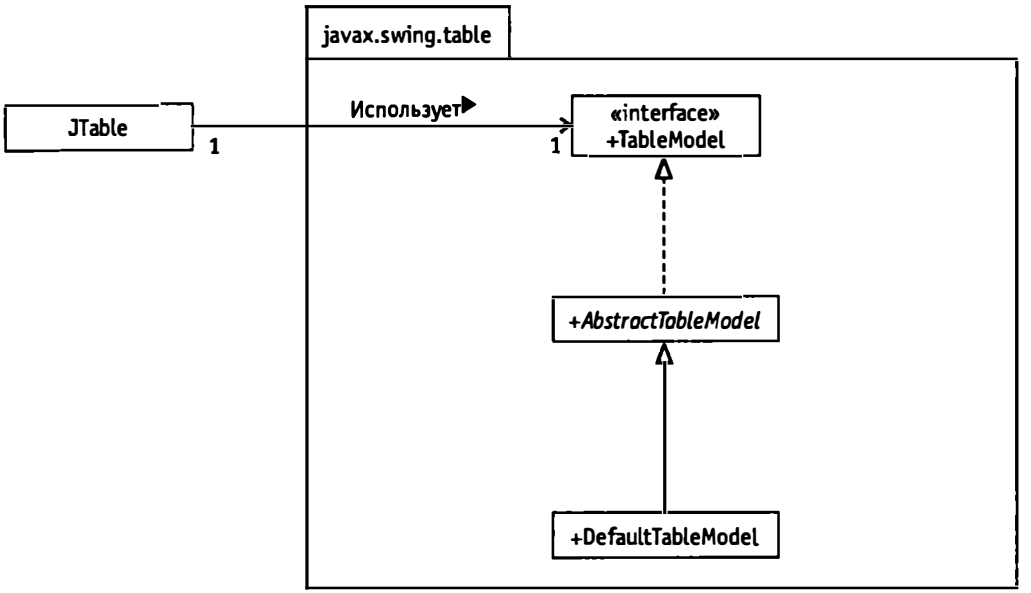


Рис. 4.12. Отношения внутри интерфейса javax.swing.table

ПРИМЕР КОДА

Пример кода для шаблона Interface and Abstract Class состоит из интерфейса и классов, предназначенных для управления структурой данных, которая называется *двусвязный список* (doubly linked list). Класс DoubleLinkedListMgr выполняет различные операции вставки и удаления элементов двусвязного списка. Класс DoubleLinkedListMgr не требует того, чтобы объекты из двусвязного списка были экземплярами какого-либо определенного класса. Он только требует, чтобы все элементы реализовывали интерфейс DoubleLinkIF. Существует абстрактный класс AbstractDoubleLink, который реализует интерфейс DoubleLinkIF. Расширение класса AbstractDoubleLink — это удобный способ написания конкретных классов, которые могут обрабатываться в двусвязном списке.

Эти классы и интерфейс входят в пакет org.clickblocks.dataStructure ПО ClickBlocks, находящегося на сайте этой книги.

Двусвязный список

Двусвязный список представляет собой структуру данных, представленную в виде некоторой последовательности, в которой каждый элемент содержит ссылку на последующий и предшествующий элемент. На рис. 4.13 представлен пример такой структуры.

Преимущество двусвязного списка над массивом заключается в количестве операций, которые нужны для вставки и удаления объектов. При вставке или удалении объектов массива нужно сдвигать все содержимое массива, начиная с места, куда вставлен или откуда удален элемент. Чем больше массив, тем больше в среднем времени потребуется на вставку и удаление его элемента. Вставка и удаление объектов в двусвязном списке предполагает корректировку ссылок на предшествующий и последующий элемент.

Вставка или удаление элемента из двусвязного списка всегда требует одного и того же количества времени независимо от того, сколько в нем элементов. Недостаток состоит в том, что при поиске *n*-го элемента в двусвязном списке нужно просмотреть первые *n* элементов. Чем больше *n*, тем больше требуется времени. На поиск *n*-го элемента в массиве затрачивается всегда одно и то же количество времени, независимо от размера массива.

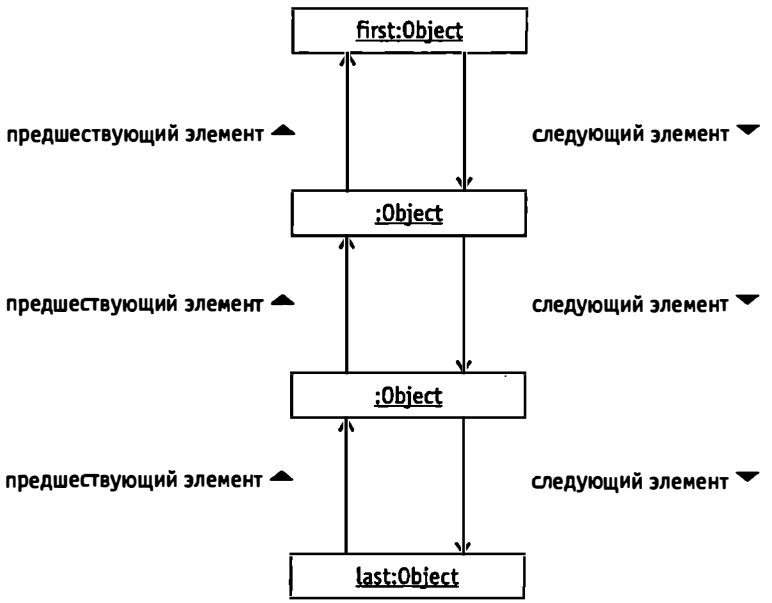


Рис. 4.13. Двусвязный список

Ниже представлен листинг интерфейса DoubleLinkIF:

```
public interface DoubleLinkIF {
   /**
    * Возвращает узел, следующий за данным узлом в связном
    * списке,или null, если данный узел — последний.
    */
   public DoubleLinkIF getNext() ;

   /**
    * Задает узел, который должен следовать за данным узлом
    * в связном списке.
    * @param node
    *     Узел, который должен следовать за данным узлом
    *     в связном списке, или null,
    *     если этот узел — последний в списке.
    */
   public void setNext(DoubleLinkIF newValue) ;

   /**
    * Возвращает узел, который должен предшествовать данному у
    * в связном списке, или null, если это первый узел.
    */
   public DoubleLinkIF getPrev() ;

   /**
    * Задает узел, который должен предшествовать данному узлу
    * в связном списке.
    *
    * @param node
    *     Узел, который должен предшествовать данному узлу
    *     в связном списке, или null,
    *     если этот узел — первый в списке.
    */
   public void: setPrev(DoubleLinkIF newValue) ;
} // interface DoubleLinkIF
```

А теперь — листинг абстрактного класса AbstractDoubleLink, который лизует интерфейс DoubleLinkIF.

```
public abstract class AbstractDoubleLink
                implements DoubleLinkIF (
   private DoubleLinkIF previous;
   private DoubleLinkIF next;
```

```
    /**
     * Возвращает узел, следующий за данным узлом в связном
     * списке, или null, если узел – последний.
     */
    public DoubleLinkIF getNext() { return next; }

    /**
     * Задает узел, который должен следовать за данным узлом
     * в связном списке.
     *
     * @param node
     *     Узел, который должен следовать за данным узлом
     *     в связном списке, или null, если этот узел –
     *     последний в списке.
     */
    public void setNext(DoubleLinkIF newValue) {
       next = newValue;
    } // setNext(DoubleLinkIF)

    /**
     * Возвращает узел, который предшествует данному узлу,
     * или null, если это первый узел.
     */
    public DoubleLinkIF getPrev() { return previous; }

    /**
     * Задает узел, который должен предшествовать данному узлу.
     *
     * @param node
     *     Узел, который должен предшествовать данному узлу
     *     в связном списке, или null, если этот узел –
     *     первый в списке.
     */
    public void setPrev(DoubleLinkIF newValue) {
       previous = newValue;
    } // setPrev(DoubleLinkIF)
...
} // class AbstractDoubleLink
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ INTERFACE AND ABSTRACT CLASS

Interface. Шаблон Interface and Abstract Class использует шаблон Interface.

Abstract Superclass. Шаблон Interface and Abstract Class использует шаблон Abstract Superclass.

Immutable (Неизменный)

Шаблон Immutable является основным не потому, что его используют многие остальные шаблоны, а потому, что чем чаще его применяют в подходящих местах кода, тем более надежными и управляемыми становятся программы.

СИНОПСИС

Шаблон Immutable повышает надежность объектов, которые совместно используют ссылки на один и тот же объект, и уменьшает затраты на параллельный доступ к объекту. Это достигается путем наложения запрета на изменение содержимого совместно используемого объекта после того, как объект уже был создан. Кроме того, шаблон Immutable не нуждается в синхронизации потоков при конкурентном доступе к одному и тому же объекту.

КОНТЕКСТ

Объекты значений (value objects) — это объекты, основное назначение которых скорее заключается в том, чтобы инкапсулировать значения, а не в том, чтобы определять поведение. Например, в классе `java.awt.Rectangle` инкапсулированы данные о расположении и размерах прямоугольника.

Если множество объектов имеет общий доступ к одному и тому же объекту значений, то проблема возникает тогда, когда процессы общего объекта значений не скоординированы надлежащим образом между объектами, имеющими к нему совместный доступ. Такая координация требует внимательности во время программирования, так как очень велика вероятность ошибок. Если изменение состояния и получение данных о состоянии общих объектов производятся асинхронно, то для надлежащего функционирования программы должна учитываться не только большая вероятность ошибок, но также и дополнительные затраты на синхронизацию доступа к состоянию общего объекта.

Шаблон Immutable позволяет избежать таких проблем. Он объявляет класс таким образом, что информация о состоянии экземпляров этого класса никогда не меняется после их создания.

Допустим, проектируется игра с участием многих игроков. Программа включает размещение и случайное перемещение объектов по игровому полю. При проектировании классов для этой программы определяется, что для представления позиции объектов на игровом поле нужно использовать неизменные объекты. Структура класса, моделирующего позицию объектов на игровом поле, представлена на рис. 4.14.

Класс `Position` имеет переменные x и y, связанные с его экземплярами. В этом классе объявлен конструктор, который устанавливает значения переменных

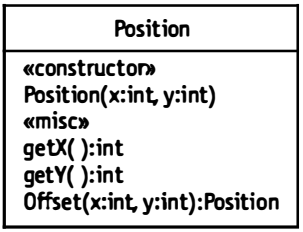


Рис. 4.14. Неизменная позиция

x и *y*. Класс также содержит методы для считывания значений *x* и *y*, связанных с его экземплярами. И наконец, он имеет метод, который создает новый объект `Position`. Параметры метода `offset` задают смещение *x* и *y* по отношению к существующему расположению на игровом поле. Класс не имеет каких-либо методов, изменяющих его переменные *x* и *y*. В случае изменения позиции объекта нужно будет создать новый объект `Position`.

МОТИВЫ

☺ Программа использует пассивные по своей сути экземпляры класса, которым даже не нужно изменять собственное состояние. Экземпляры такого класса используются многими другими объектами.

☺ Координация изменений содержимого объекта значений, используемого многими другими объектами, может быть причиной появления ошибок. Если содержимое объекта значений меняется, то все объекты, которые его используют, должны быть проинформированы об этом. Кроме того, если несколько объектов используют один объект, то они могут попытаться изменить его состояние недопустимыми способами.

☺ Если несколько потоков одновременно пытаются изменить содержимое объекта значений, то операции, выполняющие такие изменения, должны быть синхронизированы, чтобы не нарушить непротиворечивость содержимого. Издержки, связанные с синхронизацией потоков, могут усложнить доступ к содержимому объекта значений.

☺ Альтернативой изменению содержимого объекта значений является замена всего объекта другим объектом, имеющим другое содержимое.

☹ Если содержимое объекта изменяется многочисленными потоками, то замена объекта другим (вместо обновления его содержимого) не исключает необходимости синхронизации потоков, выполняющих обновление.

☹ Замена объекта значений обновленным предусматривает копирование неизменяемых значений из прежнего объекта в новый. Если изменения объекта происходят часто или если объект обладает очень большим количеством информации, цена замены объектов значений может оказаться непомерно высокой.

РЕШЕНИЕ

Чтобы не нужно было управлять синхронизацией изменения объектов значений, используемых многими другими объектами, делают общие объекты неизменными, запрещая любые изменения их состояния после их создания. Это реализуется следующим образом: в классах этих объектов не объявляются никакие методы (за исключением конструкторов), которые изменяют информацию о состоянии. Структура такого класса представлена на рис. 4.15.

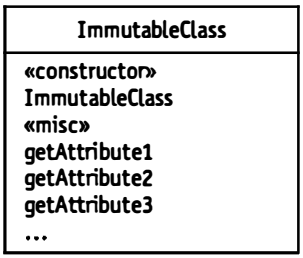


Рис. 4.15. Пример неизменяемого класса

Заметим, что класс имеет методы доступа для считывания информации о состоянии, но не для ее записи.

РЕАЛИЗАЦИЯ

Существуют два момента, о которых нужно знать при реализации шаблона Immutable:

- никакой другой метод, кроме конструктора, не должен изменять значения переменных экземпляра класса;
- любой метод, который получает информацию о новом состоянии объекта, должен сохранять эту информацию в новом экземпляре того же класса, а не изменять состояние существующего объекта.

Одно небольшое замечание, связанное с реализацией шаблона Immutable: обычно в этом шаблоне не содержится объявлений переменных с модификатором `final`. Значения результирующих переменных экземпляра обычно задаются в рамках своего класса. Однако значения переменных экземпляра неизменного объекта предоставляются другим классом, инстанциирующим объект.

СЛЕДСТВИЯ

☺ Поскольку состояние неизменных объектов никогда не изменяется, нет необходимости писать код для управления такими изменениями.

☺ Неизменный объект часто используется в качестве значения атрибута другого [illegible] необходимость синхронизации доступа к такому

атрибуту может не возникнуть. Причина заключается в том, что язык Java гарантирует, что присваивание объектной ссылки переменной всегда выполняется как атомарная операция. Если значение переменной — ссылка на объект и один поток обновляет значение объекта, а другие потоки считывают это значение, то они будут считывать либо новую, либо старую объектную ссылку.

☹ Операции, которые должны были бы изменять состояние объекта, создают новый объект. Для изменяемых объектов такие затраты не характерны.

ПРИМЕНЕНИЕ В JAVA API

Экземпляры класса `String` являются неизменными. Последовательность символов, описываемая объектом `String`, определяется при его создании. Класс `String` не содержит каких-либо методов для изменения последовательности символов, представленной объектом `String`. Методы класса `String`, такие как `toLowerCase` и `substring`, вычисляют новую последовательность символов и возвращают ее в новом объекте `String`.

ПРИМЕР КОДА

Ниже приводится примерный код для класса `Position`, описанного в разделе «Контекст»:

```
class Position {
   private int x;
   private int y;

   public Position(int x, int y) {
      this.x = x;
      this.y = y;
   } // constructor(int, int)

   public int getX() { return x; }

   public int getY() { return y; }

   public Position offset(int xOffset, Int yOffset) {
      return new Position(x+xOffset, y+yOffset);
   } // offset(int, int)
} // class Position
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ IMMUTABLE

Single Threaded Execution. Шаблон Single Threaded Execution чаще всего используется для координирования многопоточного доступа к совместно используемому объекту. Шаблон Immutable может применяться для того, чтобы избежать необходимости использования шаблона Single Threaded Execution или координации доступа любого другого вида.

Read-Only Interface. Шаблон Read-Only Interface является альтернативой шаблону Immutable. Он позволяет некоторым объектам изменять объект значений, тогда как другие объекты могут только считывать его значения.

Marker Interface (Маркер-интерфейс)

Шаблон Marker Interface редко применяется без служебных классов. Однако он включен в данную главу, так как имеет преимущество, заключающееся в фундаментальном характере объявления классов.

СИНОПСИС

Шаблон Marker Interface, чтобы показать семантические булевы атрибуты этого класса, использует тот факт, что класс реализует некоторый интерфейс. Этот шаблон особенно эффективен при использовании вместе со служебными классами, которые должны сделать какой-то вывод об объекте, ничего не зная о том, экземплярами какого определенного класса эти объекты являются.

КОНТЕКСТ

Класс `Object` в языке Java задает метод `equals`. Аргумент метода `equals` может быть ссылкой на любой объект. Поскольку класс `Object` в языке Java представляет собой исходный суперкласс для всех других классов, то все остальные классы наследуют метод `equals` от класса `Object`. Реализация метода `equals` в классе `Object` равнозначна оператору ==. Он возвращает true, если передаваемый ему объект является тем же объектом, что и он сам. Если классы хотят, чтобы их экземпляры считались равными в случае содержания в них одних и тех же значений, они соответствующим образом должны заместить метод `equals`.

Контейнерные объекты, например, `java.util.ArrayList`, вызывают метод `equals` объекта во время поиска в пределах их содержимого с целью нахождения объекта, равного данному. Такие операции поиска могут вызывать метод `equals` для каждого объекта, находящегося в контейнерных объектах. Это непродуктивно в тех случаях, когда объект, являющийся предметом поиска, принадлежит классу, в котором метод `equals` не замещен. Чтобы определить, являются ли одинаковыми два объекта, вместо обращения к реализованному в классе `Object` методу `equals` лучше воспользоваться оператором ==. Если контейнерный класс способен определить, что объект поиска принадлежит классу, в котором не замещается метод `equals`, то этот контейнерный класс может использовать оператор == вместо вызова метода `equals`. При таком подходе проблема состоит в том, что нет способа, который определял бы наличие замещения метода `equals` в произвольном классе объекта.

Существует возможность снабдить контейнерные классы подсказкой, чтобы сообщить им об уместности использования оператора == при проверке равенства объектов класса. Можно задать интерфейс с названием `EqualByIdentity`,

в котором не объявляются ни методы, ни переменные. Затем можно написать контейнерные классы и предположить, что если класс реализует `EqualByIdentity`, то равенство объектов проверяется при помощи оператора `==`.

Интерфейс, не объявляющий ни методов, ни атрибутов, но используемый для указания атрибутов классов, реализующих их, называется *маркером-интерфейсом.*

МОТИВЫ

☺ Служебные классы иногда должны иметь некоторую информацию о предполагаемом использовании класса объекта (т.е. true или false), не полагаясь на то, что объект является экземпляром определенного класса.

☺ Классы могут реализовывать любое количество интерфейсов.

☺ Существует возможность определять, реализует ли класс известный интерфейс, вне зависимости от того, экземпляром какого определенного класса он является.

☹ Некоторые атрибуты, определяющие предназначение класса, могут изменяться на протяжении времени жизни класса.

РЕШЕНИЕ

Если экземпляры служебного класса должны определять, участвуют ли экземпляры другого класса в классификации, и при этом служебному классу ничего не известно о других классах, то служебный класс может определить, реализуют ли другие классы маркер-интерфейс. Маркер-интерфейс — это интерфейс, который не объявляет ни методов, ни переменных. Класс объявляется реализующим маркер-интерфейс, чтобы указать на его принадлежность к классификации, связанной с этим маркером-интерфейсом (рис. 4.16).

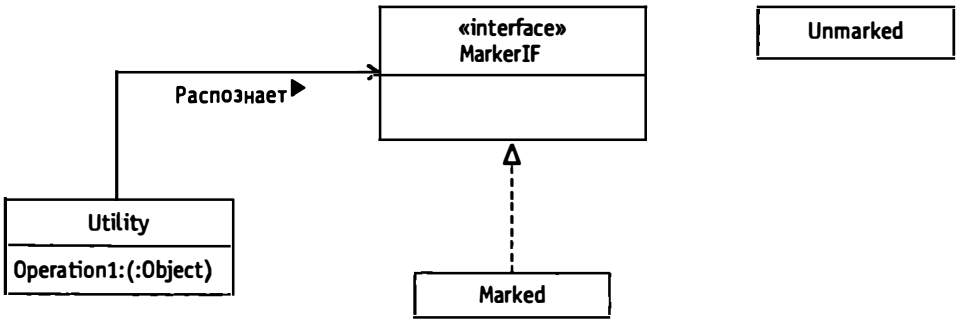


Рис. 4.16. Классы в шаблоне Marker Interface

На рис. 4.16 представлен маркер-интерфейс `MarkerIF`. На этом рисунке показан также класс `Marked`, реализующий `MarkerIF`, класс `Unmarked`, который не

реализует этот интерфейс, и служебный класс `Utility`, осведомленный об интерфейсе `MarkerIF`. Диаграмма взаимодействия между этими классами представлена на рис. 4.17.

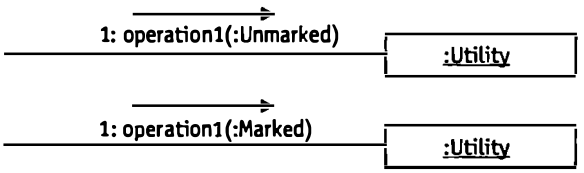


Рис. 4.17. Взаимодействие в шаблоне Marker Interface

Экземпляры класса `UtilityClass` имеют метод `operation1`. Передаваемый этому методу параметр может представлять собой объект, который реализует или не реализует интерфейс `MarkerIF`.

РЕАЛИЗАЦИЯ

Суть шаблона Marker Interface состоит в том, что объект, реализующий либо не реализующий маркер-интерфейс, передается методу служебного класса. Соответствующий такому объекту формальный параметр обычно объявляется как имеющий тип `Object`. Иногда уместно объявить такой параметр как относящийся к более конкретному классу.

Существует также возможность в шаблоне Marker Interface использовать интерфейс, в котором объявлены методы. Такой интерфейс, используемый в качестве маркера-интерфейса, обычно является наследником «чистого» маркера-интерфейса.

Если объявление класса предполагает реализацию маркера-интерфейса, то это означает участие класса в классификации, связанной с интерфейсом. При этом считается, что все подклассы такого класса тоже участвуют в классификации. Если существует хотя бы малая вероятность, что кто-то объявит подклассы, не соответствующие классификации, то необходимо принять меры по предотвращению этого. Например, объявить класс как `final` и тем самым воспрепятствовать созданию подклассов или объявить как `final` его метод `equals` и тем самым запретить его замещение.

СЛЕДСТВИЯ

☺ Экземпляры служебных классов могут делать выводы, касающиеся объектов, передаваемых их методам, вне зависимости от того, экземплярами какого определенного класса эти объекты являются.

• Отношения между служебным классом и маркером-интерфейсом являются прозрачными для всех остальных классов, за исключением классов, реализующих интерфейс.

ПРИМЕНЕНИЕ В JAVA API

Реализуя интерфейс `Serializable`, класс указывает на то, что его экземпляры могут быть сериализированы. Экземпляры класса `ObjectOutputStream` записывают объекты в виде байтовых потоков, которые экземпляр класса `ObjectInputStream` может читать и превращать обратно в объект. Преобразование объекта в поток байтов называется *сериализацией*. Существуют причины, почему экземпляры некоторых классов не должны подвергаться сериализации. Поэтому класс `ObjectOutputStream` не выполняет сериализацию тех объектов, класс которых не реализует интерфейс `Serializable`, указывающий на разрешение сериализации.

ПРИМЕР КОДА

В качестве примера использования шаблона Marker Interface рассмотрим класс, реализующий структуру данных в виде связного списка. В конце листинга находятся методы `find`, `findEq` и `findEquals`. Их назначение состоит в том, чтобы обнаружить узел `LinkedList`, ссылающийся на заданный объект. Только один из трех методов — `find` — является открытым. Метод `findEq` выполняет необходимые проверки равенства при помощи оператора ==, а метод `findEquals` — при помощи метода `equals` объекта, ради которого проводится поиск. Метод `find` решает, какой из двух методов (`findEq` или `findEquals`) должен быть вызван. Для этого он определяет, реализует ли объект, ради которого проводится поиск, маркер-интерфейс `EqualByIdentity`.

```
public class LinkedList implements Cloneable, java.io.Serializable {
...
   /**
    * Найти в связном списке объект, равный данному объекту.
    * Обычно равенство определяется путем вызова метода equals
    * данного объекта. Но если данный объект реализует интерфейс
    * EqualByIdentity, то равенство
    * будет определяться с помощью оператора ==.
    */
   public LinkedList find(Object target) {
      if (target == null || target instanceof EqualByIdentity)
         return findEq(target);
      else
         return findEquals(target);
   } // find(Object)

   /**
    * Найти в связном списке объект, равный данному объекту.
    * Равенство определяется с помощью оператора ==.
```

```
    private synchronized LinkedList findEq(Object target) {
...
    } // find(Object)

    /**
     * Найти в связном списке объект, равный данному объекту.
     * Равенство определяется путем вызова метода
     * equals данного объекта.
     */
    private synchronized LinkedList findEquals(Object target) {
...
    } // find(Object)
} // class LinkedList
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ MARKER INTERFACE

Snapshot. Шаблон Marker Interface является частью шаблона Snapshot и используется для определения возможности сериализации объектов.

Polymorphism. Шаблон Polymorphism (описанный в книге [Grand99]) представляет собой альтернативный способ изменения поведения при вызове метода.

Proxy (Заместитель)

Proxy — очень распространенный шаблон, используемый многими другими шаблонами, но никогда не применяющийся сам по себе. Шаблон Proxy был описан ранее в работе [GoF95].

СИНОПСИС

Шаблон Proxy заставляет обращаться к объекту косвенно. Обращение к методам этого объекта делегируется через объект-заместитель. Здесь имеется в виду, что запросы к реальному объекту идут через объект-заместитель. Классы для объектов-заместителей объявляются таким образом, что клиентские объекты не знают, что они имеют дело с заместителем.

КОНТЕКСТ

Объект-заместитель — это объект, методы которого вызываются на правах другого объекта. Клиентские объекты вызывают методы объекта-заместителя, которые выполняют действия, ожидаемые клиентами, не напрямую. Они вызывают методы объекта, который обеспечивает реальный сервис (рис. 4.18).

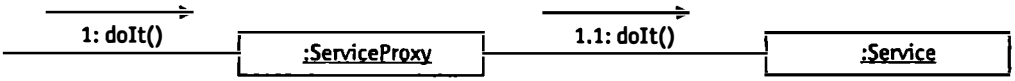


Рис. 4.18. Обращения к методам через объект-заместитель

Хотя методы объекта-заместителя не прямым образом обеспечивают сервис, ожидаемый клиентами, сам объект-заместитель обеспечивает некоторое управление такими сервисами. Объекты-заместители вместе с объектами, предоставляющими сервис, используют общий интерфейс. Имеют ли клиентские объекты прямой доступ к объектам, предоставляющим сервис, или они обращаются к объекту-заместителю — в любом случае они осуществляют доступ скорее через общий интерфейс, чем через экземпляр некоторого конкретного класса. Поэтому клиентские объекты могут не знать о том, что они вызывают методы объекта-заместителя, а не объекта, реально обеспечивающего сервис. Прозрачное управление сервисными функциями другого объекта — это основная причина использования объекта-заместителя.

Объект-заместитель может использоваться для управления сервисом самых разных видов. Некоторые наиболее важные описаны в этой книге как отдельные шаблоны. Ниже приводятся самые распространенные случаи использования

- Заместитель создает видимость немедленного возврата метода, выполнение которого требует длительного времени.
- Заместитель создает иллюзию того, что объект, на самом деле находящийся на другом компьютере, — это обычный локальный объект. Заместитель такого рода называется *удаленным заместителем* (remote proxy) или *заглушкой* (stub), и его используют RMI (Remote Method Invocation, удаленный вызов метода), CORBA (Common Object Request Broker Architecture, технология построения распределенных приложений) и другие ORB (Object Request Brokers, объектный брокер запросов). Описание классов-заглушек входит в описание ORB (в книге [Grand2001]).
- Заместитель контролирует доступ к объекту, обеспечивающему сервис, с точки зрения безопасности. Подобное использование заместителей описано как шаблон Protection Proxy (в книге [Grand2001]).
- Заместитель создает иллюзию существования объекта, обеспечивающего сервис, еще до его реального создания. Это может быть выгодно в том случае, когда создание объекта, предоставляющего сервис, требует больших затрат, а этот сервис может не понадобиться. Данный случай применения заместителей описан как шаблон Virtual Proxy (см. гл. 7).

МОТИВЫ

☺ Обеспечивающий сервис объект не может предоставлять сервис в то время и в том месте, где это удобно.

☺ Доступ к объекту, предоставляющему сервис, должен контролироваться без дополнительного усложнения сервис-объекта или привязки сервиса к политике контроля доступа.

☺ Управление сервисом должно строиться так, чтобы оно было максимально прозрачным для клиентов этого сервиса.

☺ Клиенты объекта, предоставляющего сервис, не должны заботиться о характере класса этого объекта или о том, с каким экземпляром этого класса они работают.

РЕШЕНИЕ

Весь доступ к сервис-объекту должен осуществляться через объект-заместитель. Чтобы обработка обращений была прозрачной для клиентов, объект-заместитель и сервис-объект должны либо быть экземплярами общего суперкласса, либо реализовывать общий интерфейс (рис. 4.19).

На диаграмме представлена структура шаблона Proxy, но не показаны детали, имеющие отношение к реализации политики управления доступом. Однако шаблон Proxy не очень полезен, если он не реализует определенную политику

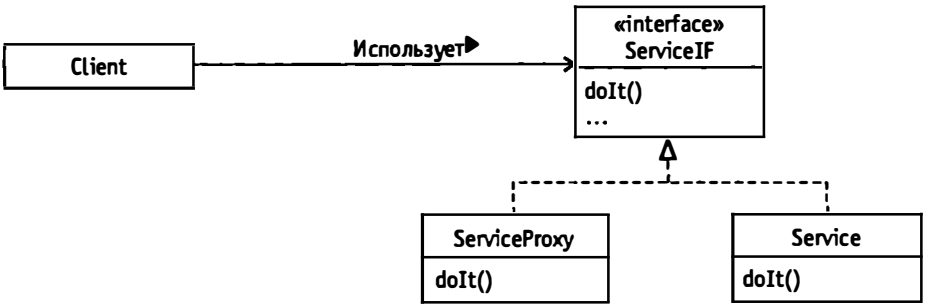


Рис. 4.19. Классы в шаблоне Proxy

управления доступом к сервис-объекту. Шаблон Proxy настолько широко используется вместе с реализацией управления доступом, что эти структуры в некоторых книгах описываются как отдельные шаблоны.

РЕАЛИЗАЦИЯ

Без реализации управления доступом к сервис-объекту реализация шаблона Proxy предполагает только создание класса, который имеет общий суперкласс или интерфейс с сервис-классом и делегирование операции экземплярам сервис-класса.

СЛЕДСТВИЯ

☺ Способ управления сервисом, предоставляемым сервис-объектом, прозрачен для объекта и его клиентов.

☺ Если использование заместителей не приводит к возникновению новых видов исключений, то, как правило, нет необходимости в том, чтобы в коде клиентских классов учитывать использование заместителей.

ПРИМЕР КОДА

Шаблон Proxy обычно используется в сочетании с логикой управления доступом к сервис-объекту. В приведенном далее примере для шаблона Proxy заместители используются с целью отложить требующую больших затрат операцию до тех пор, пока она действительно не понадобится. Если операция не потребуется, то она никогда не будет выполняться.

Рассмотрим пример класса-заместителя `java.util.HashMap`, который реализует интерфейс `java.util.Map` и `java.long.Cloneable`. Задача заместителя состоит в том, чтобы отложить клонирование базового объекта `Map` до тех пор, пока не станет ясно, что эта требующая больших затрат операция действительно нужна.

Clone

Все классы в языке Java наследуют метод `clone` от класса `java.lang.Object`. Метод `clone` объекта возвращает поверхностную копию объекта. Объект, возвращаемый методом `clone`, является экземпляром того же самого класса, что и первоначальный объект. Все его переменные экземпляра имеют те же значения, которые имеет первоначальный объект. Переменные экземпляра копии ссылаются на тот же самый объект, на который ссылается первоначальный объект.

В принципе, возможность копировать объект, содержащий секретную информацию, — это просчет в системе безопасности. Поэтому метод `clone`, унаследованный от класса `Object`, генерирует исключение, если класс объекта не разрешает выполнять клонирование. Класс допускает клонирование своих экземпляров, если он реализует интерфейс `java.lang.Cloneable`.

Многие классы, которые разрешают клонирование своих экземпляров, замещают метод `clone`, чтобы избежать таких ситуаций, когда их экземпляры совместно используют некоторые объекты, но не должны этого делать.

Одна из наиболее распространенных причин клонирования объекта `Map` заключается в том, чтобы избежать блокировки объекта в течение длительного времени, когда единственной целью является считывание множества пар «ключ — значение». Если программа многопоточная и необходимо обеспечить согласованное состояние такого объекта `Map` во время выборки из него пар «ключ — значение», то можно использовать синхронизацию для получения эксклюзивного доступа к этому объекту. В это время другие потоки будут ожидать получения доступа к тому же объекту `Map`. Подобное ожидание может быть неприемлемым.

Не каждый класс, реализующий интерфейс `Map`, допускает клонирование своих экземпляров. Тем не менее многие классы `Map`, например, `java.util.HashMap` и `java.util.TreeMap`, действительно разрешают клонирование своих экземпляров.

Клонирование объекта `Map` перед выборкой его значений — это защитная мера, которая позволяет избежать необходимости синхронизации объекта `Hashtable` сверх того времени, которое требуется для завершения операции клонирования. Если есть вновь созданная копия объекта `Map`, можно быть уверенным, что никакой другой поток не имеет доступа к этой копии. Поэтому вы можете считывать пары «ключ — значение» из копии, и другие потоки вам не помеха.

Если после клонирования объекта `Map` не производилось никаких изменений первоначального объекта `Map`, то время и память, затраченные на создание клона, были потеряны. Цель этого примера — показать, как избежать таких потерь.

Решение заключается в том, чтобы отложить клонирование объекта Map до тех пор, пока не произойдет реальное изменение объекта.

Имя класса-заместителя — LazyCloneMap. Его экземпляры представляют собой копии скопированного при записи заместителя для объекта Map. Когда вызывается метод clone заместителя, он возвращает копию заместителя, но не копирует базовый объект Map. В этот момент как оригинал, так и копия заместителя ссылаются на один и тот же базовый объект Map. Когда один из заместителей получит запрос на изменение базового объекта Map, он определит, что совместно использует общий базовый объект Map, и перед тем, как изменить его, выполнит клонирование базового объекта Map. На рис. 4.20 представлена структура класса LazyCloneMap.

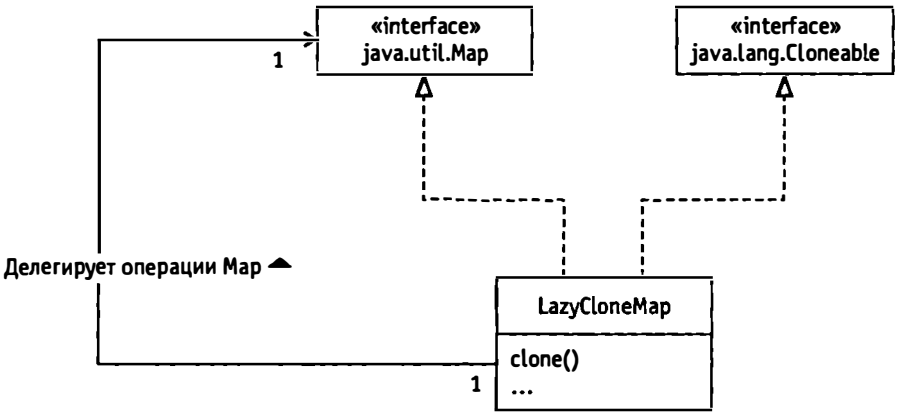


Рис. 4.20. Класс-заместитель LazyCloneMap

Начало исходного текста класса LazyCloneMap выглядит так:

```
public class LazyCloneMap implements Map, Cloneable {
   /**
    * Объект Map, для которого этот объект является заместителем.
    */
   private Map underlyingMap;
```

Классы-заместители дают возможность объекту-заместителю ссылаться на объект, для которого он функционирует как заместитель. Этот класс-заместитель имеет переменную экземпляра, которая ссылается на базовый объект Map. Объекты LazyCloneMap знают, когда они совместно используют общий базовый объект Map вместе с другими объектами LazyCloneMap, с помощью счетчика, вычисляющего количество ссылок на один и тот же базовый объект Map. Значение счетчика содержится в экземпляре класса MutableInteger. Объект MutableInteger совместно используется теми же объектами LazyCloneMap, которые совместно используют базовый объект Map.

```
/**
 * Это количество объектов-заместителей,
 * которые совместно используют один и тот же базовый объект.
 */
private MutableInteger refCount;
```

Метод clone, который классы наследуют от класса Object, является защищенным[1]. Не существует интерфейса, реализованного классами HashMap и TreeMap, который объявлял бы метод clone открытым. Поэтому класс LazyCloneMap должен получить доступ к методу clone базовых объектов Map, явно используя java.lang.reflect.Method.

```
/**
 * Переменная используется для вызова метода clone
 * базового объекта Map.
 */
private Method cloneMethod;

private static Class[] cloneParams = new Class[0];

/**
 * Constructor
 *
 * @param underlyingMap
 *     Объект Map, для которого этот объект должен быть
 *     заместителем.
 * @throws NoSuchMethodException
 *     Если базовый объект Map не содержит открытого метода
 *     clone.
 * @throws InvocationTargetException
 *     Объект, созданный этим конструктором, использует клон
 *     данного объекта Map. Если метод clone объекта Map
 *     генерирует исключение, этот конструктор генерирует
 *     InvocationTargetException, чей метод getCause возвращает
 *     исключение, сгенерированное методом clone базового
 *     объекта Map.
 */
public LazyCloneMap(Map underlyingMap)
                    throws NoSuchMethodException,
                    InvocationTargetException {
```

[1] В Java 1.4 он уже является открытым. (*Примеч. ред.*)

```
        Class mapClass = underlyingMap.getClass();
        cloneMethod = mapClass.getMethod("clone",cloneParams);
        try {
           this.underlyingMap =
              (Map)cloneMethod.invoke(underlyingMap, null);
        } catch (IllegalAccessException e) {
           // Этого не должно случиться.
        } // try
        refCount = new MutableInteger(1);
     } // constructor(Map)
```

Этот метод `clone` класса не копирует базовый объект `Map`. Он наращивает счетчик ссылок, который совместно используется первоначальным объектом `LazyCloneMap` и копией. Значение счетчика (больше единицы) говорит объектам `LazyCloneMap` о том, что они совместно используют общий объект `Map`.

```
public Object clone() {
   LazyCloneMap theClone;
   try {
      Cloneable original = (Cloneable)underlyingMap;
      theClone = (LazyCloneMap)super.clone();
   } catch (CloneNotSupportedException e) {
      // Этого не должно быть никогда.
      theClone = null;
   } // try
   refCount.setValue(1+refCount.getValue());
   return theClone;
} // clone()
```

Следующий закрытый метод вызывается открытыми методами класса `LazyCloneMap`, например, `put` и `clear`, которые изменяют базовый объект `Map`. Открытые методы вызывают этот закрытый метод перед изменением базового объекта `Map`.

```
private void ensureUnderlyingMapNotShared() {
   if (refCount.getValue()>1) {
      try {
         underlyingMap =
            (Map) cloneMethod. invoke (underlyingMap, null);
         refCount.setValue(refCount.getValue()-1);
         refCount = new MutableInteger(1);
      } catch (IllegalAccessException e) {
```

```
            // Этого не должно случиться.
        } catch (InvocationTargetException e) {
            Throwable cause = e.getCause();
            throw new RuntimeException("clone failed",
                                       cause);
        } // try
    } // if
} // ensureUnderlyingMapNotShared()
```

Сначала метод `ensureUnderlyingMapNotShared` определяет, больше ли единицы значение счетчика ссылок. Если оно больше единицы, метод знает, что этот объект `LazyCloneMap` совместно использует свой базовый объект `Map` вместе с другими объектами `LazyCloneMap`. Метод `ensureUnderlyingMapNotShared` клонирует базовый объект `Map`, и поэтому данный объект `LazyCloneMap` будет иметь свою собственную копию базового объекта `Map`, которая уже не является совместно используемой. Он уменьшает счетчик ссылок, поэтому другие объекты `LazyCloneMap`, с которыми этот объект совместно использовал базовый объект `Map`, будут знать, что они больше не разделяют один и тот же общий объект `Map` с этим объектом. Кроме того, он создает новый объект, содержащий счетчик ссылок со значением, равным 1. Это говорит о том, что данный объект `LazyCloneMap` не участвует в совместном использовании базового объекта `Map`.

Остальные методы класса `LazyCloneMap` делегируют свои функции соответствующему методу базового объекта `Map`.

```
public int size(){
    return underlyingMap.size();
}

public boolean isEmpty(){
    return underlyingMap.isEmpty();
}

public boolean containsKey(Object key){
    return underlyingMap.containsKey(key);
}

public boolean containsValue(Object value){
    return underlyingMap.containsValue(value);
}

public Object get(Object key){
    return underlyingMap.get(key);
}
```

Следующие четыре метода изменяют базовый объект Map, поэтому перед тем, как делегировать свои функции базовому объекту Map, они вызывают метод ensureUnderlyingMapNotShared.

```
public Object put(Object key, Object value){
   ensureUnderlyingMapNotShared();
   return underlyingMap.put(key, value);
}

public Object remove(Object key){
   ensureUnderlyingMapNotShared();
   return underlyingMap.remove(key);
}

public void putAll(Map m){
   ensureUnderlyingMapNotShared();
   underlyingMap.putAll(m);
}

public void clear(){
   ensureUnderlyingMapNotShared();
   underlyingMap.clear();
}

public Set keySet(){
   return underlyingMap.keySet();
}

public Collection values(){
   return underlyingMap.values();
}

public Set entrySet(){
   return underlyingMap.entrySet();
}

public boolean equals(Object that){
   return underlyingMap.equals(that);
}

public int hashCode(){
   return underlyingMap.hashCode();
}
```

Листинг класса `MutableInteger`, который использует класс `LazyCloneMap`:

```
public class MutableInteger {
   public int val;

   public MutableInteger( int value ) {
      setValue( value );
   }

   public int getValue() {
      return( val );
   }

   public void setValue( int value ) {
      val = value;
   }
   ...
}
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ PROXY

Protection Proxy. Шаблон Protection Proxy (описанный в книге [Grand2001]) использует заместителя для проведения политики безопасности при доступе к сервис-объекту.

Facade. Шаблон Facade использует одиночный объект как внешний интерфейс скорее для набора взаимосвязанных объектов, чем для единственного объекта.

Object Request Broker. Шаблон Object Request Broker (описанный в книге [Grand2001]) использует заместителя для сокрытия того факта, что сервис-объект находится не на том компьютере, на котором находятся клиентские объекты, которые хотят его использовать.

Virtual Proxy. Этот шаблон использует заместителя для создания иллюзии существования сервис-объекта еще до того, как он создается в действительности. Это удобно в том случае, когда создание объекта требует больших затрат, а его сервисы могут не понадобиться. Заместитель, рассмотренный в разделе «Пример кода» для шаблона Proxy, — это разновидность виртуального заместителя.

Decorator. С точки зрения структуры шаблон Decorator аналогичен шаблону Proxy в том смысле, что он активизирует доступ к сервис-объекту, который должен быть осуществлен косвенно через другой объект. Отличие состоит в том, какая цель при этом преследуется. Вместо попытки управления сервисом объект, благодаря которому осуществляется косвенность, некоторым образом расширяет возможности сервиса.

ГЛАВА

Порождающие шаблоны проектирования

Порождающие шаблоны предоставляют руководство к действию — как создавать объекты, если их создание требует принятия решений такого рода: экземпляр какого класса должен создаваться или каким объектам будет делегироваться ответственность за создание экземпляров. Значение порождающих шаблонов заключается в том, чтобы проинформировать нас, как структурировать и инкапсулировать эти решения.

Чаще всего в какой-то ситуации можно использовать несколько порождающих шаблонов. Иногда их выгодно комбинировать. В других случаях следует выбрать один из подходящих шаблонов. Поэтому важно ознакомиться со всеми шаблонами, представленными в данной главе.

Если у вас есть время на изучение только одного шаблона данной главы, то рассмотрите шаблон Factory Method, так как он самый распространенный. Шаблон Factory Method дает возможность объекту инициировать создание другого объекта, ничего не зная о классе этого объекта.

Шаблон Abstract Factory позволяет объектам инициировать создание объектов самых разных видов, ничего не зная о классах создаваемых объектов, но гарантируя надлежащую согласованность этих классов.

Шаблон Builder разрешает определить класс создаваемого объекта по его содер-

Шаблон Prototype дает возможность объекту создавать специальные объекты, не имея точной информации об их классе или о деталях их создания.

Шаблон Singleton позволяет многочисленным объектам совместно использовать общий объект, не зная о том, существует ли он.

При помощи шаблона Object Pool можно не создавать новые объекты, а многократно использовать старые.

Factory Method (Метод фабрики)

Этот шаблон был ранее описан в работе [GoF95].

СИНОПСИС

Нужно создать объект, представляющий внешние данные, или обработать какое-то внешнее событие. Тип объекта зависит от содержимого внешних данных или от типа события. Необходимо, чтобы ни источник данных, ни источник события, ни клиенты события не были осведомлены о реальном типе создаваемого объекта. В таком случае инкапсулируют решение о том, какой класс объекта создавать.

КОНТЕКСТ

Рассмотрим проблему написания каркаса приложений для обработки текста. Такие приложения обычно создаются для работы с документами. Их функционирование обычно начинается с команды создания или редактирования текстового документа, таблицы, плана или любого документа, с которым должно работать приложение.

Каркас такого приложения включает в себя поддержку общих операций (например, создание, открытие или сохранение документов). Эта поддержка обычно заключается в вызове определенной последовательности методов, когда пользователь задает команду. Назовем класс, предоставляющий эти методы, — `Application`.

Логика, лежащая в основе реализации большинства этих команд, меняется в зависимости от типа документа, поэтому класс `Application` обычно делегирует большинство команд некоторому объекту документа. Однако существуют операции (например, отображение заголовка документа), общие для всех объектов документа. Все сказанное выше подразумевает создание следующей структуры:

- интерфейс документа;
- абстрактный класс, который реализует общую для конкретных классов документов логику;
- конкретные, зависящие от типа документа классы, которые реализуют интерфейс для документов специального типа.

Подобная структура представлена на рис. 5.1.

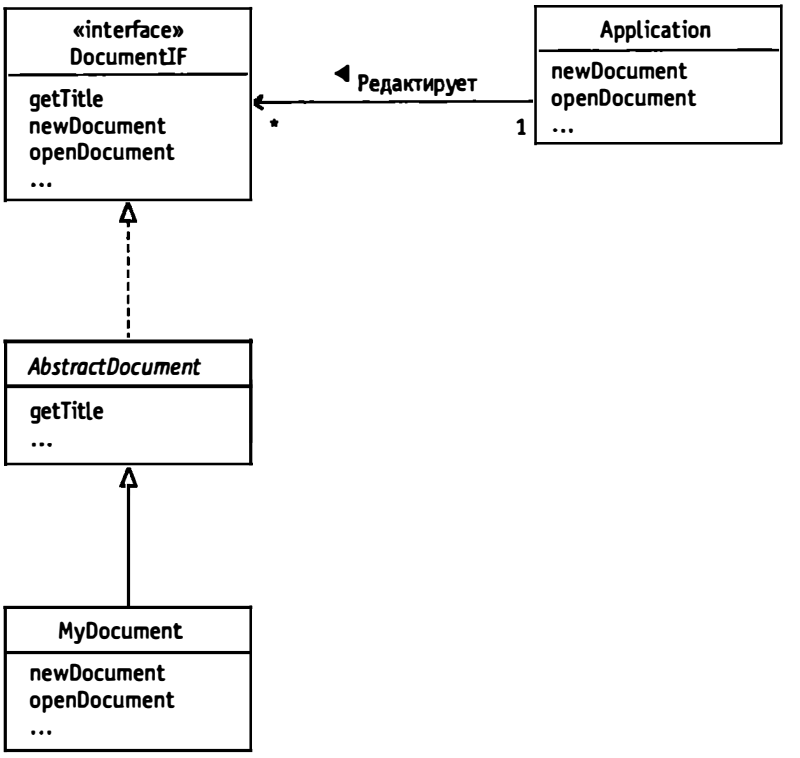


Рис. 5.1. Каркас приложения

Ни на рис. 5.1, ни в ходе предыдущего рассмотрения не было показано, как независимый от различных типов документов объект `Application` создает экземпляры классов различных типов документов.

Программист может решить данную проблему посредством использования каркаса для создания класса, инкапсулирующего логику, предназначенную для выбора и инстанциирования зависящих от типа документов классов. Чтобы класс `Application` мог вызывать созданный программистом класс, будучи независимым от него, каркас приложения должен предоставлять интерфейс, который имплементирует созданный программистом класс. Такой интерфейс объявлял бы метод, предназначенный для выбора и инстанциирования класса. Класс `Application` будет работать через интерфейс, предоставляемый каркасом приложения, а не с классом, предоставляемым программистом. На рис. 5.2 оказана такая структура.

Объект `Application` вызывает метод `createDocument` объекта, реализующего интерфейс `DocumentFactoryIF`. Он передает строку методу `createDocument`, которая сообщает, какой подкласс класса `Document` инстанциировать. Классу `pplication` не нужно знать о реальном классе объекта, метод которого он ызывает, или о том, какие подклассы класса `Document` он инстанциирует.

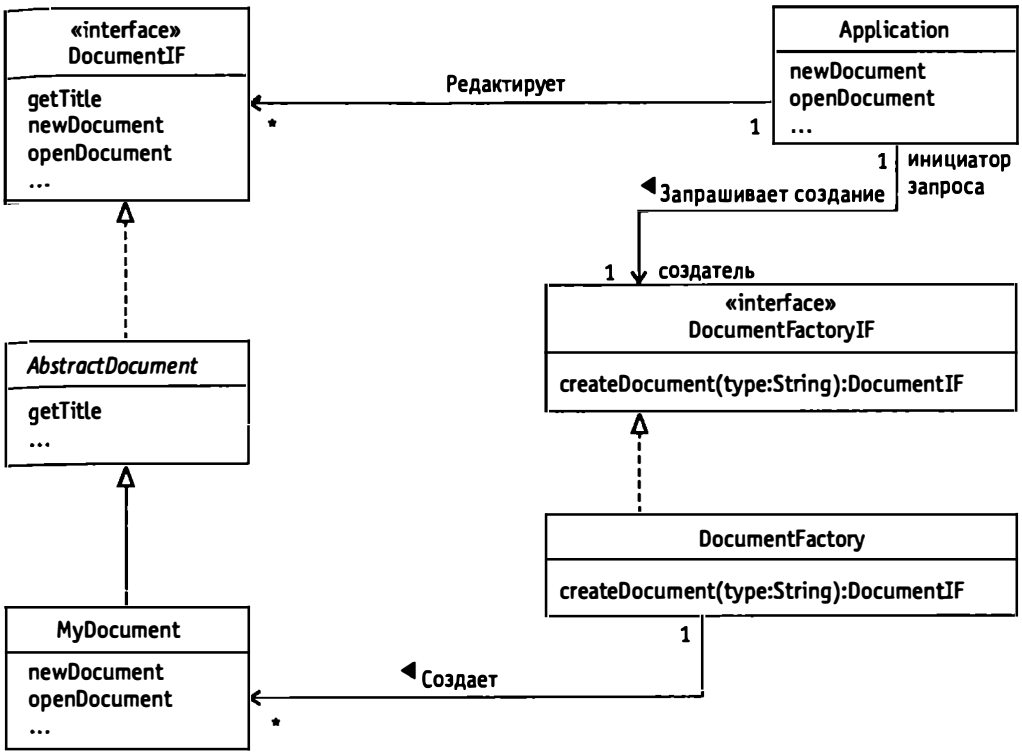


Рис. 5.2. Каркас приложения вместе с классом-фабрикой документов

МОТИВЫ

☺ Класс должен иметь возможность инициировать создание объектов, не будучи каким-либо образом зависимым от класса создаваемого объекта.

☺ Считается, что класс может инстанциировать набор классов, который может быть динамичным и изменяться по мере того, как становятся доступными новые классы.

РЕШЕНИЕ

Шаблон Factory Method представляет собой независимые от приложения объекты вместе с зависимым от приложения объектом, которому они делегируют создание других, зависимых от приложения объектов. При этом требуется присутствие независимых от приложения объектов, которые инициируют операцию создания зависимых от приложения объектов, с предположением, что эти объекты реализуют общий интерфейс.

На рис. 5.3 показаны интерфейсы и классы, которые обычно образуют шаблон Factory Method.

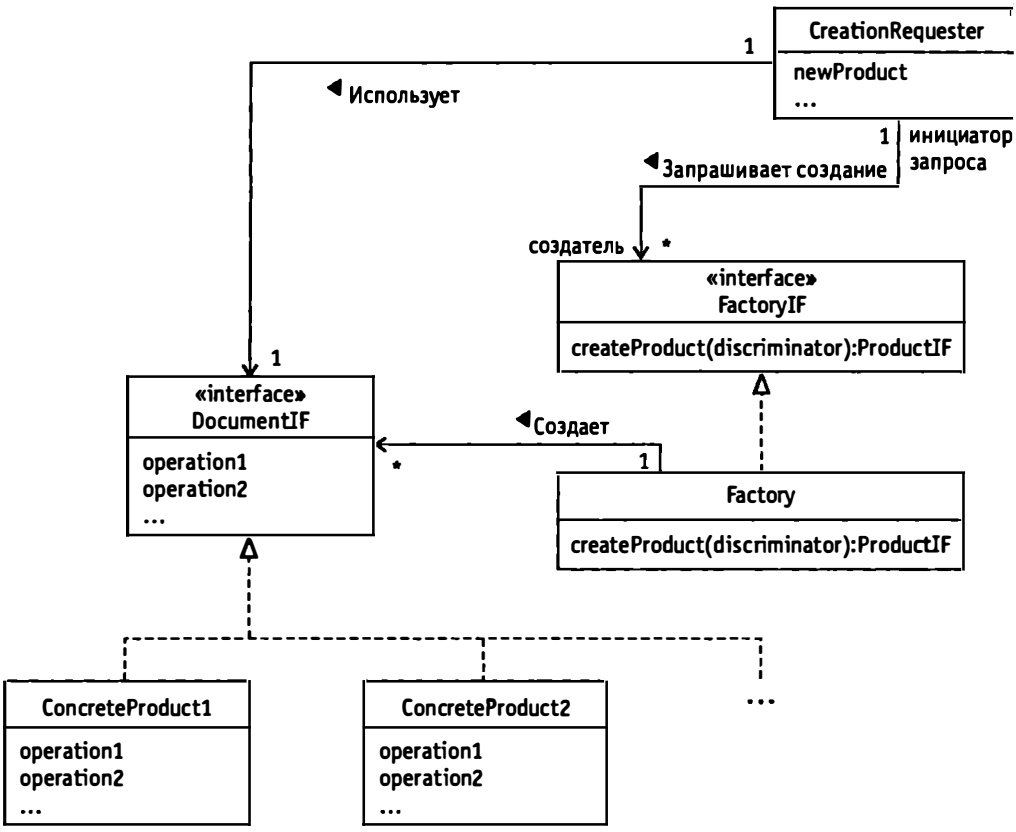


Рис. 5.3. Шаблон Factory Method

Рассмотрим роли, которые играют эти классы и интерфейсы.

ProductIF. Объекты, созданные с помощью шаблона Factory Method, должны реализовывать интерфейс, выступающий в этой роли.

Concrete Product1, Concrete Product2 и т.д. Классы, играющие эти роли, инстанциируются объектом `Factory`. Эти классы должны реализовывать интерфейс `ProductIF`.

Creation Requester. Класс, выступающий в этой роли, не зависит от приложения и нужен для создания классов, зависящих от приложений. Он делает это косвенно, при помощи экземпляра класса, реализующего интерфейс `FactoryIF`.

Factory IF. Это не зависящий от приложения интерфейс. Объекты, которые создают объекты `ProductIF` на правах объектов `CreationRequester`, должны реализовывать этот интерфейс. Интерфейсы такого рода объявляют метод, который вызывается объектом `CreationRequester` для создания объектов конкретных продуктов. Аргументы, которые принимает этот метод, рассматриваются в разделе «Реализация» в описании данного шаблона.

Выполняющие эту роль интерфейсы обычно носят имя, которое включает слово «Factory», например, `DocumentFactoryIF` или `ImageFactoryIF`.

Factory. Это зависящий от приложения класс, который реализует соответствующий интерфейс `FactoryIF` и имеет метод для создания объектов `ConcreteProduct`. Выполняющие эту роль классы обычно имеют имена, включающие слово «Factory», например, `DocumentFactory` или `ImageFactory`.

РЕАЛИЗАЦИЯ

Во многих реализациях шаблона Factory Method классы `ConcreteProduct` не прямым образом реализуют интерфейс `ProductIF`. Вместо этого они наследуются от абстрактного класса, который реализует этот интерфейс. Причины такого поведения рассматриваются при описании шаблона Interface and Abstract Class (см. гл. 4).

Определение класса по конфигурации

Существуют два основных варианта шаблона Factory Method. В общем случае класс создаваемого объекта определяется на стадии создания этого объекта. Менее общий случай предполагает, что класс создаваемого объекта всегда один и тот же и определяется до начала создания объекта.

Программа может использовать объект-фабрику, который всегда создает объект одного и того же класса, если этот класс задается некоторыми параметрами конфигурации. Предположим, например, что компания продает систему торговых терминалов, которая должна поддерживать связь с удаленным компьютером с целью обработки оплаты по кредитным карточкам. Ожидается, что классы этой системы будут отправлять сообщения удаленному компьютеру и получать ответы, используя объекты, которые реализуют определенный интерфейс. Точный формат отправляемых сообщений будет зависеть от компании, обрабатывающей операции, производимые по кредитным карточкам. Для каждой такой компании выделен соответствующий класс, который реализует нужный интерфейс и знает, как отправлять сообщения, ожидаемые компанией. Имеется также соответствующий класс-фабрика. Подобная структура показана на рис. 5.4.

Когда начинает работать система терминалов, она считывает соответствующую информацию по конфигурации, где сообщается, что она должна передать обработку операции по кредитной карте либо `BancOneCC`, либо `WellsCCFargo`. На основании этой информации система торговых терминалов создает либо объект `BancOneCCFactory`, либо `WellsCCFactory` и получает к нему доступ через интерфейс `CreditCardProcessorFactoryIF`. Когда объект должен обрабатывать операцию с кредитной картой, он вызывает метод `createProcessor` своего интерфейса `CreditCardProcessorFactoryIF`, который и создает объект, использующий заданную в конфигурации процедуру обработки кредитной карты. Обратите внимание, что метод `Create` класса-фабрики не нуждается в каких-либо аргументах, поскольку он всегда возвращает объект одного и того же типа.

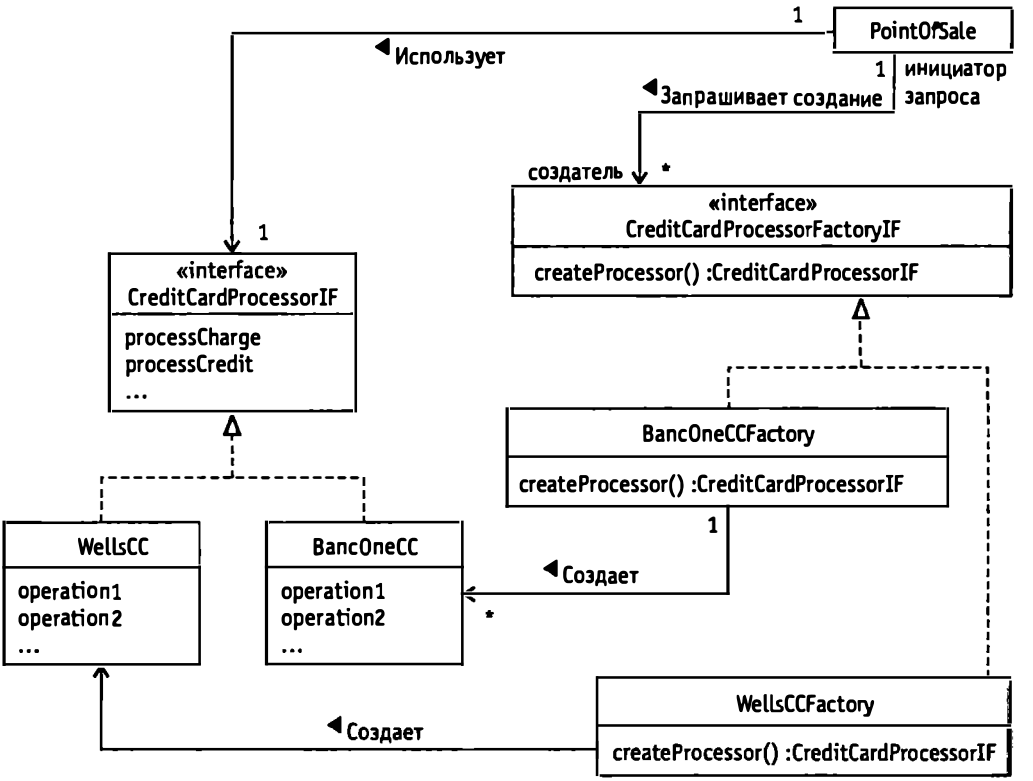


Рис. 5.4. Обработка кредитной карты

Определение класса на основе данных

Очень часто класс объекта, создаваемого объектом-фабрикой, определяется на основании данных, которые должны быть в нем инкапсулированы. Класс определяется методом `createProduct` объекта-фабрики. Определение обычно основывается на информации, переданной методу в виде параметра. Метод `createProduct` часто имеет следующий вид:

```
Image createImage (String ext) {
   if (ext.equals("gif")
      return new GIFImage();
   if (ext.equals("jpeg"))
      return new JPEGImage();
   ...
} // createImage(String)
```

Такой набор команд `if` хорошо работает для тех методов `createProduct`, которые имеют постоянный набор классов продуктов, подлежащих инстанциированию.

Чтобы написать метод `createProduct`, обрабатывающий переменное или большое число классов продуктов, можно использовать шаблон Hashed Adapter Object (описанный в книге [Grand99]). В качестве альтернативного варианта можно использовать различные объекты, которые указывают на инстанциируемый класс, являясь ключами в хэш-таблице, значения в которой задаются объектами `java.lang.reflect.Constructor`. Применяя этот способ, ищут значение аргумента в хэш-таблице и затем используют объект конструктора, найденный в хэш-таблице, для инстанциирования нужного объекта.

Предыдущий пример иллюстрирует то, что методы объекта-фабрики — подходящее место для размещения команд `switch` или цепочки команд `if`. Во многих случаях наличие в коде команд `switch` или цепочек команд `if` указывает на то, что метод должен быть реализован как полиморфный. Методы объекта-фабрики, однако, не могут быть реализованы при помощи полиморфизма, поскольку полиморфизм работает только после того, как объект уже был создан.

Для многих случаев реализации шаблона Factory Method аргументы для метода `createProduct` объекта-фабрики могут быть представлены в виде набора заранее определенных значений. Классу-фабрике часто удобно задавать символические имена для каждого из этих заранее определенных значений. Классы, которые просят класс-фабрику создать объект, могут использовать константы, определяющие символические имена, которые задают тип создаваемого объекта.

Иногда существует многоуровневое определение класса на основе данных. Например, может существовать объект-фабрика высшего уровня, который отвечает за создание объекта-фабрики, создающего реальный объект продукта. По этой причине ту форму шаблона Factory Method, которая основана на данных, иногда называют *многоуровневой инициализацией*.

СЛЕДСТВИЯ

☺ Запрашивающий создание класс не зависит от реально создаваемых классов объектов конкретных продуктов.

☺ Набор инстанциируемых классов продуктов может изменяться динамически.

☹ Косвенность, существующая между инициированием создания объекта и определением инстанциируемого класса, может усложнить понимание программы специалистами из команды поддержки.

ПРИМЕНЕНИЕ В JAVA API

Для интеграции оболочки апплета и основной программы в некоторых местах Java API используется шаблон Factory Method. Например, каждый объект `URL` имеет связанный с ним объект `URLConnection`. Можно использовать объекты `URLConnection` для чтения исходных байтов `URL`. Кроме того, объекты

URLConnection имеют метод getContent, который возвращает содержимое URL, упакованное в соответствующем объекте. Если URL, например, содержит файл gif, то метод getContent объекта URLConnection возвращает объект Image.

Этот механизм работает следующим образом: объекты URLConnection играют роль инициатора запроса создания в шаблоне Factory Method. Они делегируют функции метода getContent объекту класса ContentHandler. Это абстрактный класс, который играет роль Product и имеет информацию об управлении содержимым определенного типа. Объект URLConnection получает объект класса ContentHandler через объект класса ContentHandlerFactory. Это абстрактный класс, который участвует в шаблоне Factory Method в качестве интерфейса класса-фабрики. Класс URLConnection имеет также метод setContentHandlerFactory. Программы, в которых выполняются апплеты, вызывают этот метод для предоставления объекта-фабрики, используемого всеми объектами URLConnection.

ПРИМЕР КОДА

Предположим, создается приложение для записи в журнал событий отчетов, созданных системой торговых терминалов[1]. Каждая строка, которая появляется на ленте кассового аппарата, записывается в журнал событий. Приложение будет читать все записи журнала и генерировать итоги транзакций.

Приложение должно будет работать с терминалами разных производителей, и записи в журнал событий будут производиться в разных форматах. По этой причине необходимо создать классы, отвечающие за генерирование итогов, независимыми от форматов.

Все форматы в журнале событий состоят из последовательности записей, типы которых одинаковы. Можно сделать генерирующие итог классы независимыми от форматов файла журнала при помощи внутреннего представления журнала событий как последовательности объектов, которые будут соответствовать записям в журнале. Для этого можно спроектировать классы, соответствующие различным типам записей, которые появляются в журнале.

Проблема такого подхода в том, как приложение будет создавать объект, представляющий отчеты в журнале событий. В этом примере используются обе формы шаблона Factory Method.

- При чтении отчетов из журнала приложение использует объект-фабрику для создания объекта, который соответствует каждой записи в журнале. Каждый раз, когда объект-фабрика получает запрос на создание объекта, он выбирает класс для создания экземпляра, основанного на информации об отчете. Это пример объекта-фабрики, который определяет класс во время выполнения.

[1] Система торговых терминалов — это высокотехнологичные кассовые аппараты.

- Класс объекта-фабрики, который создает объекты для представления отчетов, основывается на типе журнала событий, который будет читать приложение. Класс такого объекта-фабрики основывается на конфигурационной информации, поэтому он создается другим объектом-фабрикой, когда приложение читает эту конфигурационную информацию.

Организация классов, на которых основывается код такого примера, показана на рис. 5.5. Опишем эти классы и интерфейсы.

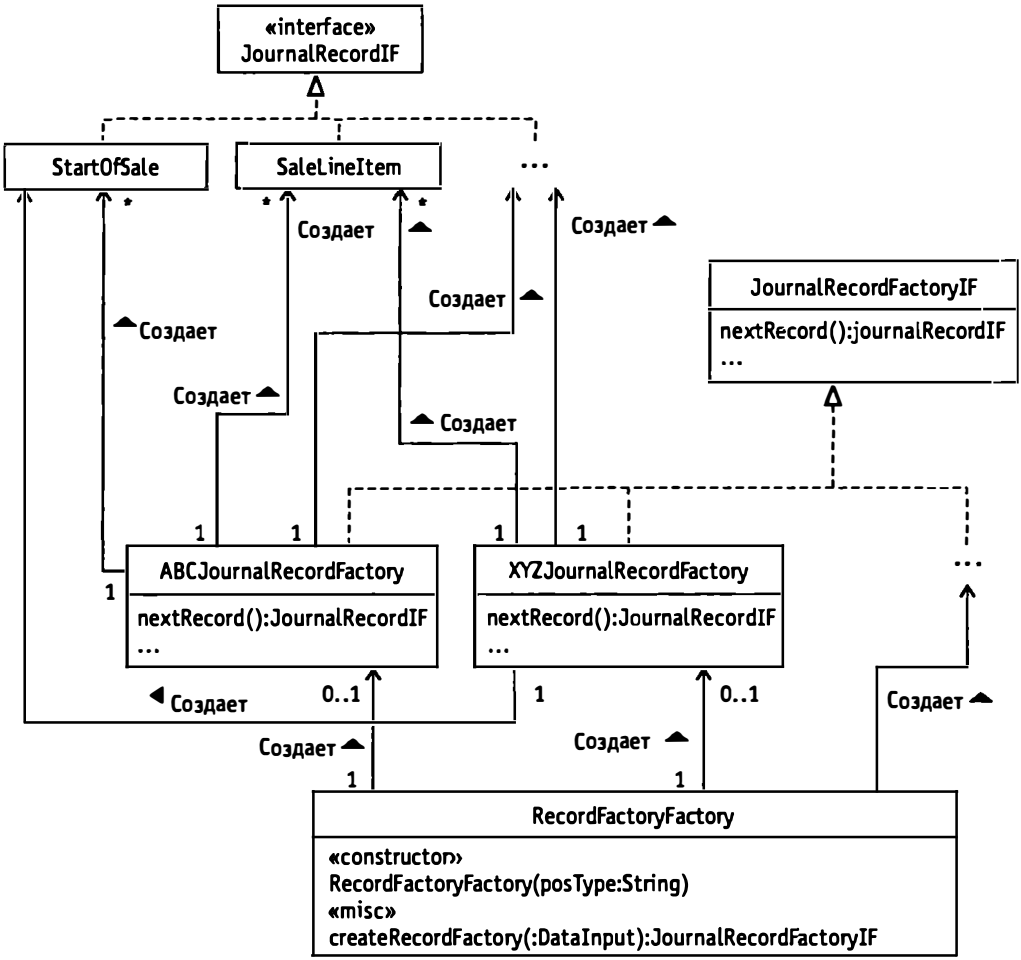


Рис. 5.5. Пример метода фабрики

JournalRecordIF. Объекты, которые инкапсулируют содержимое журнала событий, являются элементами класса, который реализует этот интерфейс.

На рис. 5.5 показаны всего два таких класса, хотя в приложении их может быть гораздо больше.

StartOfSale. Экземпляры этого класса представляют запись журнала, которая показывает начало транзакции продажи.

SaleLineItem. Экземпляры этого класса представляют запись журнала, которая содержит детали процесса продажи на кассовом аппарате.

JournalRecordFactoryIF. Этот интерфейс реализуется классами, отвечающими за сбор информации в журнале событий и ее инкапсуляцию в объектах, которые реализуют интерфейс `JournalRecordIF`. Экземпляры классов, которые реализуют этот интерфейс, создают объекты, которые реализуют интерфейс `JournalRecordIF`, например, объекты `StartOfSale` и `SaleLineItem`.

ABCJournalRecordFactory. Экземпляры этого класса отвечают за распознавание формата журнала событий для системы торговых терминалов, произведенной фирмой ABC. Экземпляры класса читают следующую запись из объекта, который реализует `java.io.DataInput`, и инкапсулируют его содержимое в экземплярах класса, который реализует интерфейс `JournalRecordIF`.

XYZJournalRecordFactory. Экземпляры этого класса отвечают за распознавание формата журнала событий для системы торговых терминалов, произведенных фирмой XYZ. Экземпляры класса читают следующую запись из объекта, который реализует `java.io.DataInput`, и инкапсулируют его содержимое в экземплярах класса, реализующих интерфейс `JournalRecordIF`.

RecordFactoryFactory. Название типа системы торговых терминалов передается конструктору этого класса. Созданный объект используется для создания экземпляров соответствующего класса, который реализует интерфейс `JournalRecordIF`, для специфического типа системы торговых терминалов.

Можно заметить структурную схожесть между этим примером, использующим две формы шаблона Factory Method, и шаблоном Abstract Factory. Одна из отличительных характеристик шаблона Abstract Factory — клиентские объекты скорее связаны с интерфейсом, реализуемым объектами, чем с их содержимым.

Рассмотрим код, реализующий приложение. Сначала — листинг класса `RecordFactoryFactory`:

```
public class RecordFactoryFactory {

    private Constructor factoryConstructor;
```

Когда создается экземпляр класса, строка, показывающая тип системы терминалов, для которой объект будет фабрикой, передается конструктору класса. Он получает объект `java.lang.reference.Constructor`, который он будет использовать для создания объектов, и делает объект `Constructor` значением экземпляра переменной `FactoryConstructor`.

```
    // POS Types
       public static final String ABC = "abc";
```

```
/**
 * Constructor
 *
 * @param posType
 * Тип системы терминалов, для которой этот объект будет
 * создавать объект JournalRecordFactoryIF.
 * @throws POSException
 * Если возникла проблема, инициализируем этот объект.
 */
public RecordFactoryFactory(String posType)
                    throws POSException {
  Class[] params = { DataInput.class };
  Class factoryClass;
  if (ABC.equals(posType)) {
    factoryClass = ABCJournalRecordFactory.class;
  ...
} else {
  String msg = "Unknown POS type: "+ posType;
  throw new java.lang.IllegalArgumentException(msg);
}
try {
  factoryConstructor
    = factoryClass.getConstructor(params);
} catch (Exception e) {
  String msg = "Error while constructing factory";
  throw new POSException(msg, e);
} // try
} // constructor(String)
```

Метод, который создает объекты JournalRecordFactoryIF:

```
public JournalRecordFactoryIF createFactory(DataInput in)
                    throws POSException {
  Object[] args = {in};
  Object factory;
  try {
    factory = factoryConstructor.newInstance(args);
  } catch (Exception e) {
    String msg = "Error creating factory";
    throw new POSException(msg, e);
  } //
  return (JournalRecordFactoryIF)factory;
} // createFactory
} // class RecordFactoryFactory
```

А теперь — интерфейс JournalRecordFactoryIF:

```
    * Этот интерфейс реализуется классами, которые
    * отвечают за создание объектов, инкапсулирующих содержимое
    * записи журнала для системы терминалов.
    */
public interface JournalRecordFactoryIF {
    /**
     * Возвращает объект, который инкапсулирует следующую
     * запись в журнале событий.
     *
     * @throws  EOFException
     *          Если в журнале событий больше нет записей.
     * @throws IOException
     *          Если при чтении следующей записи возникли
     *          какие-то проблемы.
     */
  public JournalRecordIF nextRecord() throws EOFException,
                                             IOException;

  ...
} // interface JournalRecordFactoryIF
```

Класс ABCJournalRecordFactory реализует интерфейс JournalRecordFactoryIF для типа системы терминалов, которая называется ABC.

```
public class ABCJournalRecordFactory
                    implements JournalRecordFactoryIF {
  // Типы записей.
  ...
  private static final String SALE_LINE_ITEM = "17";
  private static final String START_OF_SALE = "4";
  ...

  private DataInput in;
  private final SimpleDateFormat dateFormat
    = new SimpleDateFormat("yyyyMMddHHmmss");

  // Счетчик количества записей.
  private int sequenceNumber = 0;

  ABCJournalRecordFactory(DataInput input) {
    in = input;
  } // constructor(DataInput)
```

```
/**
* Возвращает объект, который инкапсулирует следующую запись
* в журнале событий.
*
* @throws  EOFException
*          Если в журнале событий больше нет записей.
* @throws  IOException
*          Если при чтении следующей записи возникли какие-то
*          проблемы.
*/
public JournalRecordIF nextRecord() throws EOFException,
                                           IOException {
   String record = in.readLine();
   StringTokenizer tokenizer;
   tokenizer = new StringTokenizer(record, ",");
   sequenceNumber++;

   try {
      String recordType = tokenizer.nextToken();
      ...
      if (recordType.equals(START_OF_SALE)) {
      return constructStartOfSale(tokenizer);
      } else if (recordType.eguals(SALE_LINE_ITEM)) {
      return constructSaleLineItem(tokenizer);
      ...
      } else {
         String msg = «Unknown record type»;
         throw new IOException(msg);
      } // if
   } catch (NoSuchElementException e) {
      // Это исключение рассматривается как ошибка ввода-вывода,
      // если запись не имеет все ожидаемые поля.
         String msg = "record is missing some fields";
         IOException ioe = new IOException(msg);
         ioe.initCause(e);
         throw ioe;
      } // try
} // nextRecord()

private
StartOfSale constructStartOfSale(StringTokenizer tok)
                  throws NoSuchElementException {
```

```
            String transactionID = tok.nextToken();
            tok.nextToken(); // Пропустить индикатор способа.
            String timestampString = tok.nextToken();
            Date timestamp = parseTimestamp(timestampString);
            String terminalID = tok.nextToken();

            return new StartOfSale(terminalID,
                                   sequenceNumber,
                                   timestamp,
                                   transactionID);
        } // constructStartOfSale(StringTokenizer)
        ...
    } // class ABCJournalRecordFactory
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ FACTORY METHOD

Hashed Adapter Objects. Шаблон Hashed Adapter Objects (описанный в книге [Grand99]) может использоваться при реализации шаблона Factory Method. Hashed Adapter Objects может быть полезен, если классы, которые объект-фабрика инстанциирует, могут изменяться в течение работы программы.

Abstract Factory. Шаблон Factory Method применяется при создании одиночных объектов для специфических целей без создания инициатора запроса, знающего, как специфические классы будут инстанциироваться. Если нужно создать подходящие наборы классов объектов, то больше подходит шаблон Abstract Factory.

Template Method. Шаблон Factory Method часто применяют совместно с шаблоном Template Method при определении типа того, что нужно создать с использованием конфигурационной информации.

Prototype. Шаблон Prototype позволяет объекту еще одним способом взаимодействовать с другими объектами, не зная деталей их конструкции.

Strategy. Шаблон Strategy подходит лучше, чем шаблон Template Method, для ситуаций, в которых количество вариантов поведения велико.

Abstract Factory (Абстрактная фабрика)

Abstract Factory известен также под названием Kit, или Toolkit.

Этот шаблон был ранее описан в работе [GoF95].

СИНОПСИС

Если задан набор связанных интерфейсов, то шаблон Abstract Factory предоставляет способ создания объектов, реализующих эти интерфейсы, из соответствующего набора конкретных классов. Шаблон Abstract Factory может быть полезен в том случае, когда программа должна работать с разнообразными сложными внешними реальными объектами, например, системами управления окнами, обладающими похожей функциональностью.

КОНТЕКСТ

Предположим, необходимо создать каркас пользовательского интерфейса, который должен работать со многими оконными системами, например, MS Windows, Motif или MacOS, т.е. на любой платформе с сохранением свойственного платформе внешнего вида и стиля (look and feel). Для каждого типа элемента окна (текстовое поле, кнопка, список и т.д.) создают абстрактный класс, а затем объявляют конкретный подкласс каждого такого класса для каждой поддерживаемой платформы. В целях надежности нужно гарантировать, что все созданные объекты элементов окна предназначены для нужной платформы. Здесь вступают в игру абстрактные классы-фабрики.

Абстрактный класс-фабрика определяет методы для создания экземпляра каждого абстрактного класса, представляющего элемент окна пользовательского интерфейса. Конкретные классы-фабрики представляют собой конкретные подклассы абстрактного класса-фабрики, реализующего методы для создания экземпляров конкретных классов элементов окна для одной и той же платформы.

С точки зрения общего контекста абстрактный класс-фабрика и его конкретные подклассы организуют набор конкретных классов, работающих с разными, но связанными продуктами.

Предположим, пишется программа, которая выполняет удаленную диагностику компьютеров, произведенных фирмой Stellar Microsystems. Их самые старые компьютеры использовали чипы процессоров фирмы Enginola, имеющие традиционный набор комплексных команд. С тех пор фирма разработала несколько поколений компьютеров, базирующихся на собственных архитектурах RISC с названиями ember, superember и ultraember. Основные компоненты, используемые в этих моделях, выполняют те же функции, но включают различные наборы деталей.

Чтобы создаваемая программа знала, какие тесты запускать и как интерпретировать результаты, надо инстанциировать объекты, которые будут соответствовать каждому основному компоненту диагностируемого компьютера. Класс каждого объекта будет соответствовать типу тестируемого компонента. Это означает, что будет набор классов для каждого варианта архитектуры компьютера. В каждом наборе будет класс для компонента компьютера одного и того же типа.

На рис. 5.6 представлена диаграмма классов, описывающая организацию классов, в которых инкапсулирована диагностика для компонентов различных видов. Здесь представлены только компоненты двух видов. Структура классов компонента любого другого вида будет аналогичной. Для компонента каждого типа существует интерфейс. Каждая поддерживаемая архитектура компьютера имеет класс, который реализует каждый интерфейс.

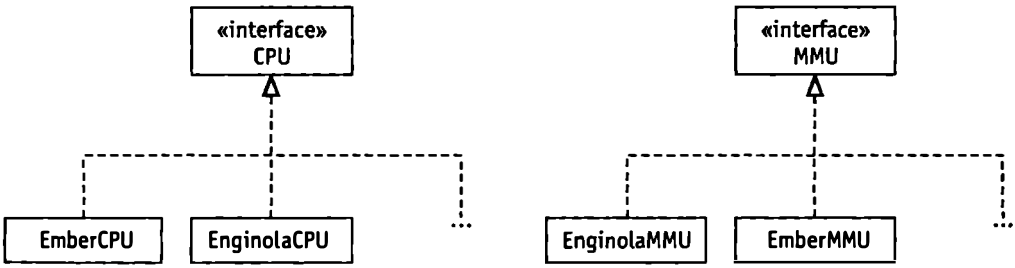


Рис. 5.6. Классы диагностики

Для решения задачи нужно знать, как организовать создание объектов диагностики системы. Желательно также, чтобы классы, использующие классы диагностики системы, не зависели бы от них. Для создания объектов диагностики системы можно было бы использовать шаблон Factory Method, но есть одна проблема, которую этот шаблон не решает: необходимо, чтобы для каждой архитектуры процессора существовали свои классы диагностики системы. Если просто использовать шаблон Factory Method, клиентские классы при запросе на создание классов диагностики системы должны будут сообщать каждому классу-фабрике, для какой архитектуры компьютера предназначены объекты диагностики системы. Нужно найти единый способ решения следующей проблемы: все объекты диагностики, используемые для компьютера, предназначены именно для той архитектуры, которая нужна, но классы, использующие эти объекты, не обременены какими-либо дополнительными зависимостями.

МОТИВЫ

☺ Система, работающая со многими продуктами, должна функционировать независимо от конкретного продукта, с которым она работает.

☺ Должна существовать возможность задавать конфигурацию системы для работы с одним или несколькими членами семейства продуктов.

- ☺ Обязательное условие: экземпляры классов, предназначенные для взаимодействия с продуктом, должны использоваться совместно и только вместе с этим продуктом.
- ☺ Остальная часть системы должна работать с продуктом, ничего не зная о конкретных классах, используемых для взаимодействия с продуктом.
- ☺ Система должна быть расширяемой, чтобы иметь возможность работать с дополнительными продуктами посредством добавления новых наборов классов при условии изменения только нескольких строк кода.
- ☹ Интерфейса, реализуемого классом, недостаточно для того, чтобы выделить его среди других классов, которые могут быть инстанциированы с целью создания некоторого объекта.

РЕШЕНИЕ

На рис. 5.7 представлена диаграмма классов, описывающая роли, исполняемые классами в шаблоне Abstract Factory.

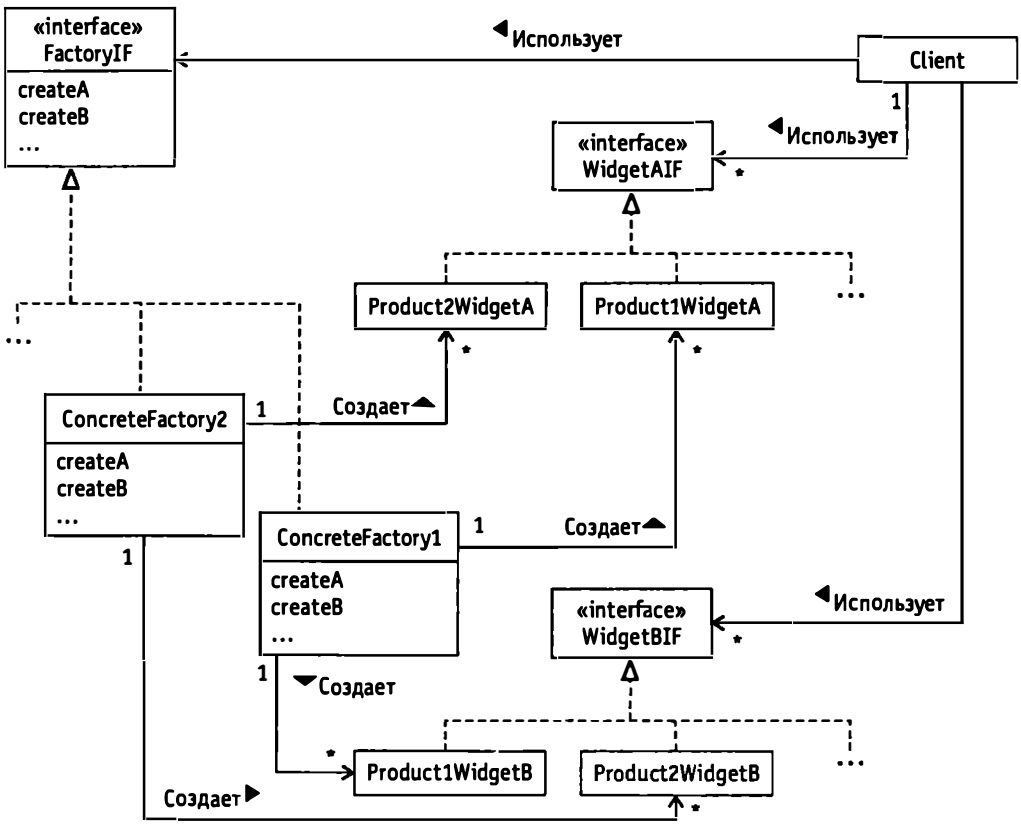


Рис. 5.7. Классы в шаблоне Abstract Factory

Рассмотрим эти роли.

WidgetAIF, WidgetBIF и т.д. Интерфейсы, выступающие в этой роли, соответствуют сервису или характеристике продукта. Классы, которые реализуют один из этих интерфейсов, работают с сервисом или характеристикой, которой соответствует интерфейс.

Ввиду нехватки места здесь показаны только два таких интерфейса. Большая часть приложений с шаблоном Abstract Factory имеет более двух интерфейсов подобного рода. Интерфейсов должно быть столько, сколько используется отдельных характеристик и сервисов продукта.

Product1WidgetA, Product2WidgetA и т.д. Классы, выступающие в этой роли, соответствуют конкретной характеристике определенного продукта. В общем, можно ссылаться на классы, играющие эти роли, как на конкретные элементы окна.

Client. Классы в роли `Client` используют конкретные классы элементов окна, чтобы запросить или получить сервис от того продукта, с которым работает клиент. Клиентские классы взаимодействуют с конкретными классами элементов окон только через интерфейс `WidgetIF`. Они не зависят от реальных конкретных классов элементов окон, с которыми работают.

FactoryIF. Интерфейсы в этой роли объявляют методы для создания экземпляров конкретных классов элементов окон. Каждый такой метод возвращает экземпляр класса, который реализует свой интерфейс `WidgetIF`. Классы, отвечающие за создание конкретных объектов элементов окон, реализуют интерфейс, выступающий в этой роли.

ConcreteFactory1, ConcreteFactory2 и т.д. Классы в этой роли реализуют интерфейс `FactoryIF` для создания экземпляров конкретных классов элементов окон. Каждый класс, выступающий в этой роли, соответствует отдельному продукту, с которым может работать класс `Client`. Каждый метод класса `ConcreteFactory` создает конкретный элемент окна определенного вида. Однако все конкретные элементы окна, создаваемые этими методами, предназначены для работы с тем продуктом, которому соответствует класс `ConcreteFactory`.

Клиентские классы, вызывающие эти методы, не должны иметь никакой непосредственной информации об этих конкретных классах-фабриках. Вместо этого они должны обращаться к экземплярам этих классов через интерфейс `FactoryIF`.

РЕАЛИЗАЦИЯ

Основной задачей, решаемой при реализации шаблона Abstract Factory, является проблема написания механизма, который используется клиентскими классами для создания и доступа к объектам `FactoryIF`. Самая простая ситуация возникает тогда, когда клиентские объекты должны работать только с одним продуктом на протяжении всего периода функционирования программы. В этом случае некоторые классы обычно имеют статические переменные, заданные для класса `ConcreteFactory`, используемого на стадии выполнения

программы. Переменная может быть открытой, или ее значение может быть получено через открытый статический метод.

Если абстрактный объект-фабрика будет использовать информацию, предоставляемую выдающим запрос клиентом, для выбора одного из нескольких конкретных объектов-фабрик, то можно использовать шаблон Factory Method.

СЛЕДСТВИЯ

☺ Клиентские классы не зависят от конкретных классов элементов окон, которые они используют.

☺ Добавление (в отличие от написания) классов для работы с дополнительными продуктами осуществляется просто. Обычно на класс конкретного объекта-фабрики ссылка должна делаться только в одном месте. Кроме того, несложно заменить конкретный объект-фабрику, используемый для работы с определенным продуктом.

☺ Заставляя клиентские классы проходить через интерфейс `FactoryIF` с целью создания конкретных объектов элементов окон, шаблон Abstract Factory гарантирует, что клиентские объекты используют соответствующий набор объектов для работы с продуктом.

☹ Главный недостаток шаблона Abstract Factory в том, что при взаимодействии с продуктом может понадобиться большая работа по написанию нового набора классов. Значительных усилий требует также расширение набора характеристик, которые существующий набор классов способен реализовать. Если должен поддерживаться новый продукт, пишется полный набор конкретных классов элементов окон с целью поддержки этого продукта. Необходимо написать конкретный класс элемента окна для каждого интерфейса `WidgetIF`. Если имеется множество интерфейсов `WidgetIF`, то для поддержки дополнительного продукта потребуется большая работа.

Добавление доступа к дополнительной характеристике взаимодействующих продуктов тоже может потребовать значительных усилий, если существует много поддерживаемых продуктов. При этом пишется новый интерфейс `WidgetIF`, соответствующий новой характеристике, и новый конкретный класс элемента окна, соответствующий каждому продукту.

☹ Клиентские объекты могут потребовать, чтобы классы элементов управления окна были организованы в виде иерархии, которая обслуживает потребности клиентских объектов. Шаблон Abstract Factory сам по себе не подходит для решения этой задачи, так как он требует организации конкретных классов элементов окон в виде иерархии классов, не зависящей от клиентских объектов. Это препятствие может быть преодолено посредством объединения шаблона Bridge с шаблоном Abstract Factory:

1. Создайте иерархию не зависящих от продуктов классов элементов окон, удовлетворяющих требованиям клиентских классов. При этом каждый не зависящий от продукта класс элемента окна делегирует

зависящую от продукта логику зависящему от продукта классу, который реализует интерфейс `WidgetIF`.

2. Пакет `java.awt` содержит классы, реализующие использование такого варианта. Классы, подобные `Button` и `TextField`, содержат логику, которая не зависит от используемой системы управления окнами. Эти классы обеспечивают естественный внешний вид и стиль посредством делегирования операций системы управления окнами конкретным классам элементов окон, реализующим интерфейсы, определенные в пакете `java.awt.peer`.

ПРИМЕНЕНИЕ В JAVA API

Шаблон Abstract Factory используется в Java API с целью реализации класса `java.awt.Toolkit`. Это класс абстрактной фабрики, применяемый для создания объектов, работающих с «родной» системой управления окнами. Используемый класс конкретной фабрики задается кодом инициализации, и единственный объект конкретной фабрики возвращается его методом `getDefaultToolkit`.

ПРИМЕР КОДА

Чтобы рассмотреть пример кода этого шаблона, вернемся к проблеме, описанной в разделе «Контекст». На рис. 5.8 представлена расширенная диаграмма классов, в которой содержится шаблон Abstract Factory.

Экземпляр класса `Client` управляет процессом удаленной диагностики. После определения архитектуры машины он должен диагностировать ее. Он передает тип архитектуры методу `createToolkit` объекта `ToolkitFactory`. Этот метод возвращает экземпляр класса, например, `EmberToolkit` или `EnginolaToolkit`, который реализует интерфейс `ArchitectureToolkitIF` для заданной архитектуры компьютера. Затем объект `Client` может использовать объект `ArchitectureToolkitIF` для создания объектов, которые моделируют CPU (Central Processing Units, центральные процессорные устройства), MMU (Memory Management Units, блоки управления памятью) и другие компоненты соответствующей архитектуры.

Приведем код, который реализует проект диагностики удаленного компьютера, изображенный на рис. 5.8. Абстрактные классы элементов окна имеют очевидную структуру.

Код для конкретного класса-фабрики, который создает экземпляры классов для тестирования компьютеров, имеющих архитектуру `Ember`, выглядит следующим образом:

```
class EmberToolkit implements ArchitectureToolkitIF{
  public CPU createCPU() {
    return new EmberCPU();
```

```
    } // createCPU()

    public MMU createMMU() {
       return new EmberMMU();
    } // createMMU()
    ...
} // class EmberFactory
```

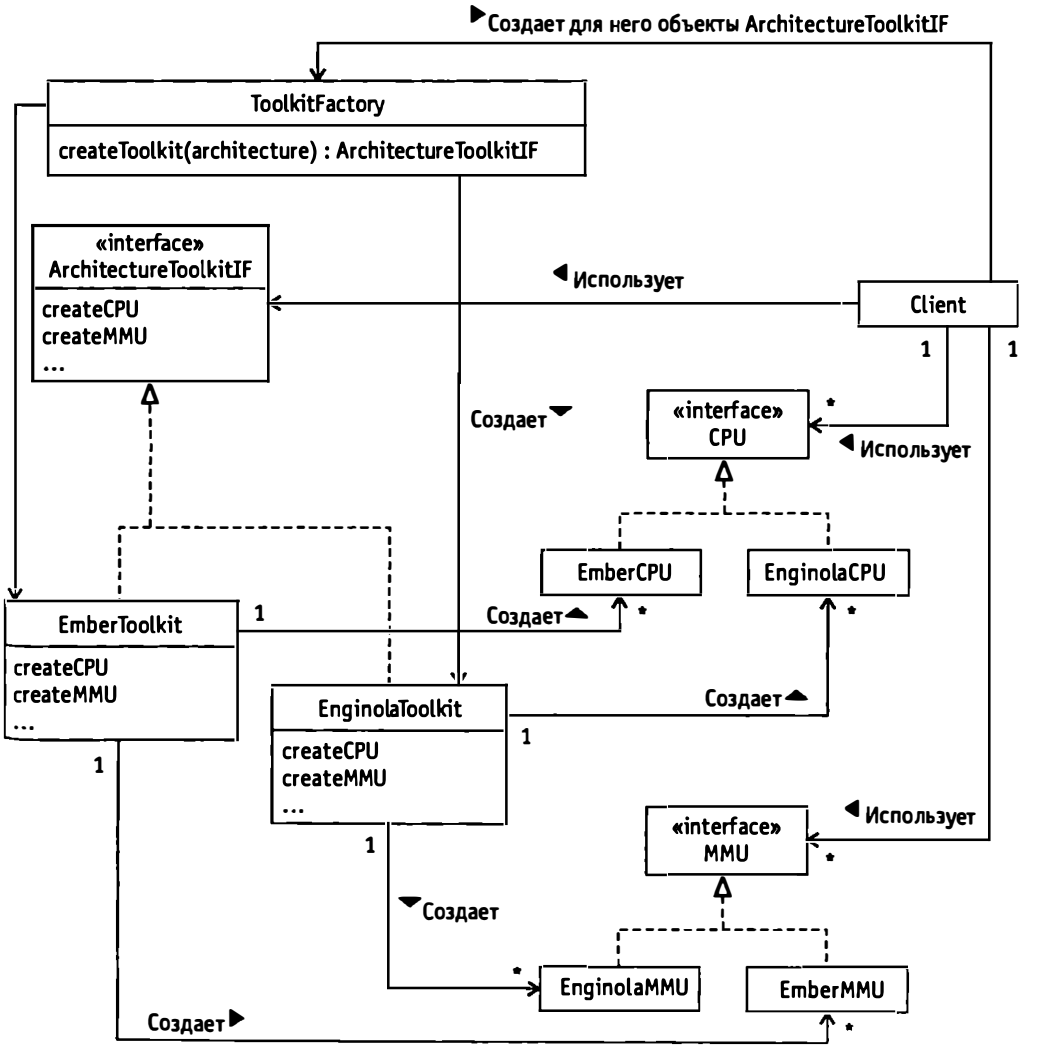


Рис. 5.8. Классы диагностики вместе с абстрактной фабрикой

Код для интерфейса фабрики:

```
public interface ArchitectureToolkitIF {
   public abstract CPU createCPU() ;
   public abstract MMU createMMU() ;
...
} // AbstractFactory
```

В этой реализации шаблона Abstract Factory для создания объектов `ArchitectureToolkitIF` используется шаблон Factory Method. В нижеприведенном листинге класс-фабрика отвечает за создание объектов `ArchitectureToolkitIF`.

```
public class ToolkitFactory {
   /**
    * Единственный экземпляр этого класса.
    */
   private static ToolkitFactory myInstance
   = new ToolkitFactory ();

   // Символические имена для идентификации архитектур компьютеров.
   public final static int ENGIHOLA = 900;
   public final static int EMBER    = 901;
   //...

   public static ToolkitFactory getInstance() {
      return myInstance;
} // getInstance()

   /**
    * Возвращает вновь созданный объект, который реализует
      интерфейс.
    * ArchitectureToolkitIF для данной архитектуры компьютера.
    */

   public
   ArchitectureToolkitIF createToolkit(int architecture) {
      switch (architecture) {
         case ENGINOLA:
            return new EnginolaToolkit();

         case EMBER:
         return new EmberToolkit();
```

```
        } // switch
        String errMsg = Integer.toString(architecture);
        throw new IllegalArgumentException(errMsg);
    } // createToolkit(int)
} // class ToolkitFactory
```

Клиентские классы обычно создают объекты конкретных элементов управления окна, используя для этой цели код, который выглядит примерно так:

```
public class Client {
    public void doIt () {
        ToolkitFactory myFactory;
        myFactory = ToolkitFactory.getInstance();
        ArchitectureToolkitIF af;
        af = myFactory.createToolkit (ToolkitFactory.EMBER);
        CPU cpu = af.createCPU();
        ...
    } // doIt
} // class Client
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ ABSTRACT FACTORY

Factory Method. В последнем примере абстрактный класс-фабрика использует шаблон Factory Method для принятия решения, какой конкретный объект-фабрику передать клиентскому классу.

Singleton. Конкретные классы-фабрики обычно реализуются как классы Singleton.

Builder (Строитель)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Builder позволяет объекту клиента создавать сложный объект, задавая для него только тип и содержимое. Клиент избавлен от информации о деталях построения объекта.

КОНТЕКСТ

Рассмотрим написание программы шлюза электронной почты. Программа получает сообщения электронной почты в формате MIME[1]. Она отправляет их в другом формате, соответствующем другому виду системы электронной почты. Эта проблема хорошо решается с помощью шаблона Builder. Очень просто создать программу, использующую объект, который занимается синтаксическим анализом сообщений в формате MIME. Каждое анализируемое сообщение во время синтаксического анализа образует пару вместе с объектом-строителем, используемым синтаксическим анализатором для создания сообщения в новом формате. Как только синтаксический анализатор распознает какой-либо заголовок и тело сообщения, он вызывает соответствующий метод объекта-строителя, с которым работает.

На рис. 5.9 изображена диаграмма классов, описывающая такую структуру.

Класс `MessageManager` отвечает за сбор сообщений электронной почты в формате MIME и активизацию их передачи. Сообщения, которыми он непосредственно управляет, являются экземплярами класса `MIMEMsg`.

Экземпляры класса `MIMEMsg` инкапсулируют сообщения электронной почты в формате MIME. Если объект `MessageManager` хочет передать сообщение в формате, отличном от MIME, то он должен заново создать сообщение в нужном формате. Содержание нового сообщения должно быть таким же, как и содержание сообщения в MIME-формате, или настолько близким, насколько позволяет новый формат.

Класс `MIMEParser` представляет собой подкласс класса `MessageParser`, который может выполнять синтаксический анализ сообщений электронной почты в формате MIME и передавать их содержание объекту-строителю.

---

[1] MIME — это аббревиатура словосочетания Multipurpose Internet Mail Extensions (многоцелевые расширения интернет-почты), определяющего стандарт, которому подчиняются почти все сообщения электронной почты интернета. Описание MIME можно найти по адресу http://mgrang.home.mindspring.com/mime.html.

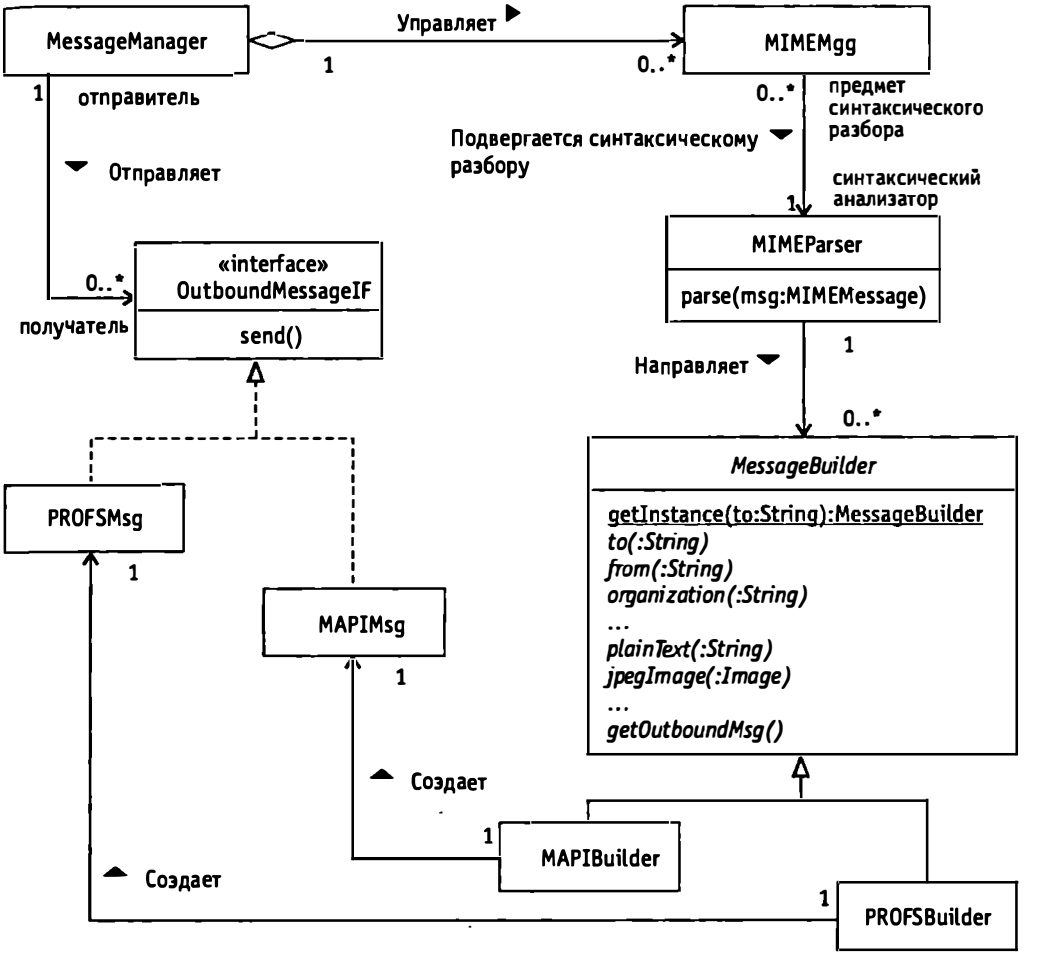


Рис. 5.9. Пример класса-строителя

`MessageBuilder` — это абстрактный класс-строитель. Он определяет методы, соответствующие различным полям заголовков и телам, которые поддерживаются стандартом MIME. Класс `MessageBuilder` объявляет абстрактные методы, которые соответствуют обязательным полям заголовков и наиболее распространенным типам сообщения. Он делает эти методы абстрактными, так как все конкретные подклассы класса `MessageBuilder` должны определять эти методы. Однако некоторые необязательные поля заголовков (например, organization) и специфические форматы сообщения (например, `Image/Jpeg`) могут не поддерживаться всеми форматами сообщений, поэтому класс `MessageBuilder` предоставляет «бездействующие» реализации таких методов.

Класс `MessageBuilder` определяет также статический метод `getInstance`, которому объект `MIMEParser` передает адрес назначения анализируемого сообщения. По этому адресу метод `getInstance` определяет формат сообщения,

необходимый для нового сообщения. Он возвращает экземпляр подкласса класса `MessageBuilder`, соответствующий формату нового сообщения, в объект `MIMEParser`.

`MAPIBuilder` и `PROFSBuilder` — это конкретные классы-строители для создания MAPI- и PROFS-сообщений соответственно.

Классы-строители создают объекты продуктов, которые реализуют интерфейс `OutboundMessageIF`. Этот интерфейс определяет метод `send`, предназначенный для отправки сообщений электронной почты по нужному адресу.

На рис. 5.10 представлена диаграмма взаимодействия, описывающая совместную работу этих классов.

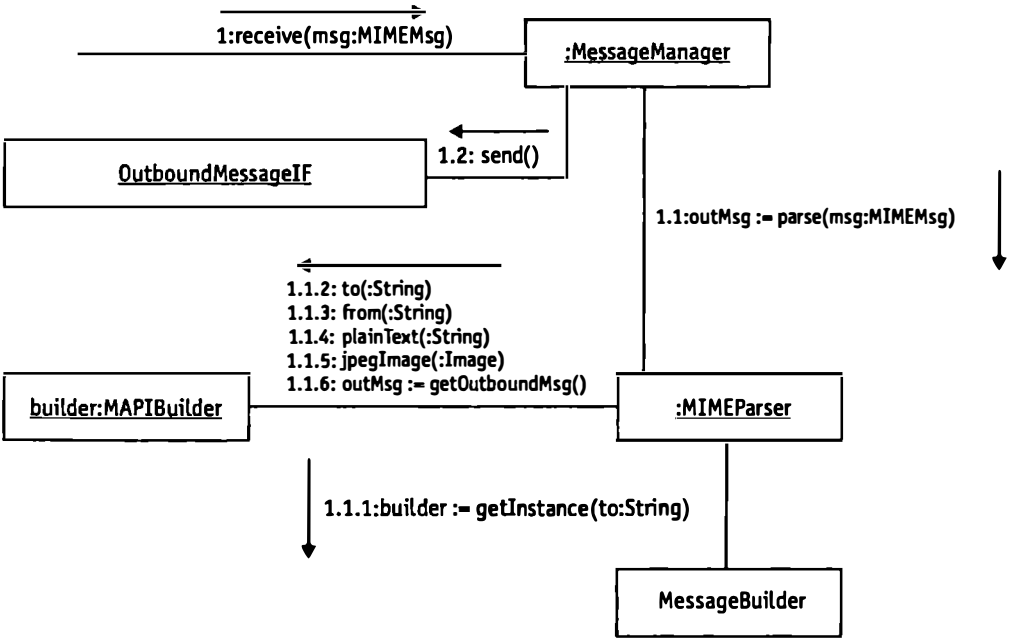


Рис. 5.10. Взаимодействие в шаблоне Builder

Опишем изображенную на рис. 5.10 ситуацию.

1. Объект `MessageManager` получает сообщение электронной почты.
   - **1.1.** Объект `MessageManager` вызывает метод `parse` класса `MIMEParser`. Он возвращает объект `OutboundMessageIF`, который инкапсулирует новое сообщение в нужном формате.
     - **1.1.1.** Объект `MIMEParser` вызывает метод `getInstance` класса `MessageBuilder`, передавая ему адрес назначения электронной почты. Анализируя адрес, метод выбирает конкретный подкласс класса `MessageBuilder` и создает его экземпляр.

1.1.2. Объект `MIMEParser` передает адрес назначения электронной почты методу `to` объекта `MessageBuilder`.

1.1.3. Объект `MIMEParser` передает исходящий адрес электронной почты методу `from` объекта `MessageBuilder`.

1.1.4. Объект `MIMEParser` передает содержимое сообщения электронной почты методу `plainText` объекта `MessageBuilder`.

1.1.5. Объект `MIMEParser` передает jpeg-изображение, присоединенное к сообщению электронной почты, методу `jpegImage` объекта `MessageBuilder`.

1.1.6. Объект `MIMEParser` вызывает метод `getOutboundMsg` объекта `MessageBuilder` для завершения обработки старого и считывания нового сообщения.

1.2. Объект `MessageManager` вызывает метод `send` интерфейса `OutboundMessageIF`. Он отправляет сообщение и завершает его обработку сообщения.

МОТИВЫ

☺ Программа должна иметь возможность создавать многочисленные внешние представления одних и тех же данных.

☺ Классы, содержащие в себе данные, не должны зависеть от какого-либо внешнего представления этих данных и от классов, участвующих в их создании. Если классы, инкапсулирующие в себе данные, не зависят от внешних представлений данных, то изменения классов, отвечающих за внешнее представление этих данных, не потребуют изменения классов, отвечающих за их содержимое.

☺ Классы, отвечающие за создание внешних представлений данных, не зависят от классов, инкапсулирующих их в себе. Их экземпляры могут работать с любым, предоставляющим содержание, объектом, ничего о нем не зная.

РЕШЕНИЕ

На рис. 5.11 представлена диаграмма классов, показывающая участников шаблона Builder.

Опишем роли, которые играют эти классы и интерфейсы в шаблоне Builder.

Product. Класс в этой роли определяет тип представления данных. Все классы `Product` должны реализовывать интерфейс `ProductIF` таким образом, чтобы другие классы могли ссылаться на объекты класса `Product` через интерфейс, ничего не зная о классе объекта.

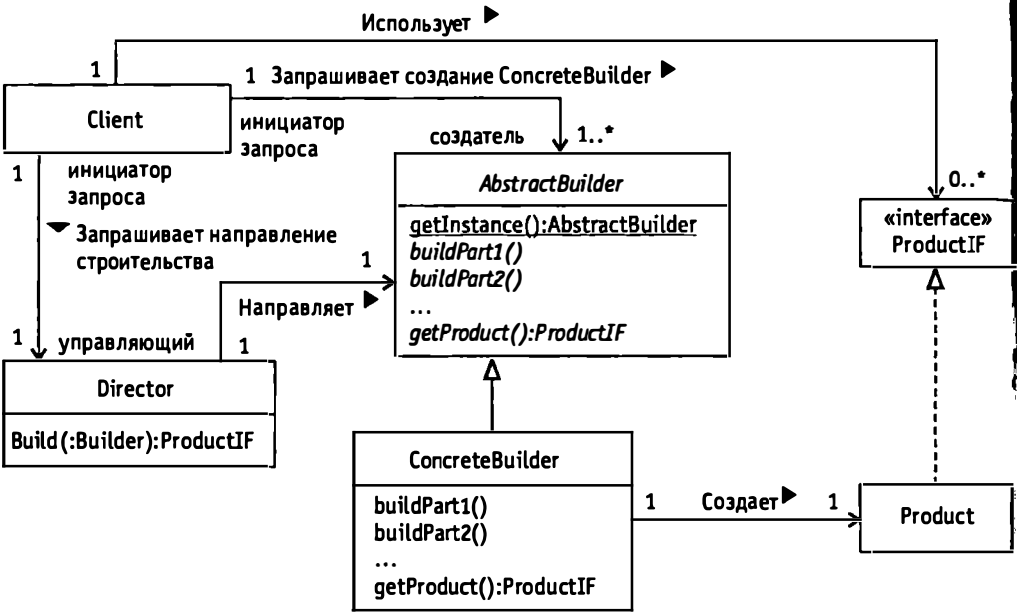


Рис. 5.11. Шаблон Builder

ProductIF. Шаблон Builder используется для создания объектов класса `Product` самых разных видов, эти объекты используются объектами класса `Client`. Чтобы объекты класса `Client` не нуждались в информации о реальном классе объектов `Product`, созданных для них, все классы `Product` реализуют интерфейс `ProductIF`. Объекты `Client` ссылаются на созданные для них объекты `Product` через интерфейс `ProductIF`, поэтому им не нужно ничего знать о реальном классе объектов, созданном для них.

Client. Экземпляр клиентского класса инициирует действия шаблона Builder. Он вызывает метод `getInstance` класса `AbstractBuilder`. Передавая информацию методу `getInstance`, он тем самым сообщает ему, продукт какого вида должен быть создан. Метод `getInstance` определяет создаваемый подкласс класса `AbstractBuilder` и возвращает его объекту `Client`. Затем объект `Client` передает объект, полученный от `getInstance`, методу `build` класса `Director`, который создает нужный объект.

ConcreteBuilder. Класс в этой роли представляет собой конкретный подкласс класса `AbstractBuilder`, который используется для создания определенного вида представления данных объектом `Director`.

AbstractBuilder. Класс в этой роли — это абстрактный суперкласс классов `ConcreteBuilder`. Класс `AbstractBuilder` имеет статический метод, обычно с именем `getInstance`, которому передается некий аргумент, задающий тип представления данных. Метод `getInstance` возвращает экземпляр класса конкретного строителя, который создает заданное представление данных.

Кроме того, класс `AbstractBuilder` определяет методы, которые представлены на диаграмме классов как `buildPart1`, `buildPart2` и т.д. Эти методы вызываются объектом `Director`, чтобы сообщить объекту, возвращаемому методом `getInstance`, какое содержимое нужно поместить в созданный объект.

И наконец, класс-строитель определяет метод, обычно с именем `getProduct`, который возвращает объект продукта, созданный конкретным объектом-строителем.

Director. Объект `Director` вызывает методы конкретного объекта-строителя, чтобы сконструировать объект продукта.

На рис. 5.12 представлена диаграмма взаимодействия, демонстрирующая, как эти классы работают вместе.

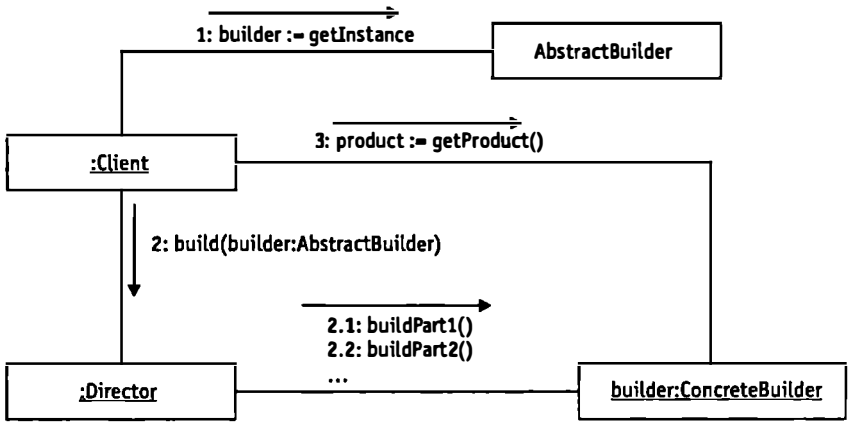


Рис. 5.12. Взаимодействие в шаблоне Builder

РЕАЛИЗАЦИЯ

В результате проектирования и реализации шаблона Builder получается набор методов, определяемых классом строителя для предоставления содержимого для конкретных объектов-строителей. Эти методы представляют основной интерес, так как их может быть очень много. Они должны быть достаточно общими, чтобы допускать создание всех приемлемых представлений данных. С другой стороны, слишком общий набор методов может вызывать больше трудностей при реализации и использовании. Противопоставление общности и сложности реализации является причиной появления следующих вопросов на стадии реализации.

1. Каждый метод, предоставляющий доступ к содержимому и объявленный в абстрактном классе-строителе, может быть абстрактным или по умолчанию быть реализован как бездействующий. Объявления абст-

рактного метода заставляют конкретные классы-строители обеспечивать реализацию этого метода. Обязывая конкретные классы-строители реализовывать такие методы, получают хорошие результаты в тех случаях, когда метод предоставляет существенную информацию о содержимом. Однако методы, которые предоставляют необязательное содержимое или дополнительную информацию о его структуре, могут быть необязательными или даже неуместными для некоторых представлений данных. При обеспечении бездействующей (по умолчанию) реализации таких методов экономятся усилия при реализации конкретных классов-строителей, которые не нуждаются в этих методах.

2. Организация конкретных классов-строителей таким образом, что обращения к предоставляющим содержимое методам просто добавляют данные в объект продукта, часто является вполне достаточной. В некоторых случаях нельзя просто сообщить классу-строителю, в каком месте конечного продукта окажется определенная часть этого продукта. В таких ситуациях, может быть, самое простое — заставить метод, предоставляющий содержимое, возвращать объект, инкапсулирующий эту часть продукта, распорядителю. Объект-распорядитель может затем передавать этот объект другому методу, предоставляющему содержимое, таким способом, который подразумевает размещение части продукта внутри целого продукта.

СЛЕДСТВИЯ

☺ Задание содержимого и построение определенного представления данных не зависят друг от друга. Представление данных продукта может изменяться, никак не влияя на объекты, предоставляющие содержимое. Объекты-строители могут работать с разными объектами, предоставляющими содержимое, не нуждаясь при этом в каких-либо изменениях.

• Шаблон Builder обеспечивает более точный контроль во время построения, чем другие шаблоны (например, Factory Method), давая возможность объекту-распорядителю шаг за шагом контролировать весь процесс создания объекта-продукта. Другие шаблоны просто создают сразу весь объект.

ПРИМЕР КОДА

Рассмотрим образец кода для классов этого примера, взаимодействующих в рамках шаблона Builder. Экземпляры класса `MIMEParser` играют роль объектов-распорядителей. Приведем исходный текст для класса `MIMEParser`:

```
class MIMEParser {
  private MIMEMessage msg;          // Синтаксически
                                    // анализируемое сообщение.
  private MessageBuilder builder;   // Объект-строитель.
```

```
    ...
    /**
     * Синтаксический анализ MIME-сообщения, в ходе которого
     * вызываются методы строителей, соответствующие полям
     * заголовков и частям тел сообщений.
     */
    OutboundMessageIF parse() {
       builder = MessageBuilder.getInstance(getDestination());
       MessagePart hdr = nextHeader();
       while (hdr != null) {
          if (hdr.getName().equals("to"))
             builder.to((String)hdr.getValue());
          else if (hdr.getName().equals("from"))
             builder.from((String)hdr.getValue());
          ...
          hdr = nextHeader();
       } // while hdr
       MessagePart bdy = nextBodyPart();
       while (bdy != null) {
          String name = bdy.getName();
          if (name.equalsIgnoreCase ("text/plain"))
             builder. plainText((String)bdy.getValue());
          ...
          else if (name.equalsIgnoreCase("image/jpeg"))
             builder.jpegImage((Image)bdy.getValue());
          ...
          bdy = nextHeader();
       } // while bdy
       return builder.getOutboundMsg();
    } // parse(Message)
...
    private class MessagePart {
          private String name;
          private Object value;

          MessagePart(String name, Object value) {
             this.name = name;
             this.value = value;
             } // Constructor(String, String)

             String getName() { return name; }

             Object getValue() { return value; }
    } // class MessagePart
} // class MIMEParser
```

Цепочки команд `if`, содержащиеся в методе `parse` показанного выше класса, будут довольно длинными, если привести полностью код этого метода. Стандарт MIME поддерживает свыше 25 видов только полей заголовков. Менее трудоемкий способ организовать цепочку проверок равенства объектов, которая в результате приводит к вызову метода, состоит в использовании шаблона Hashed Adapter Objects (описанного в книге [Grand99]).

Приведем код для класса `MessageBuilder`, который играет роль абстрактного класса-строителя:

```
abstract class MessageBuilder {
  /**
   * Возвращает экземпляр подкласса, соответствующий формату
   * сообщения электронной почты. Формат выбирается,
   * исходя из данного адреса назначения.
   * @param dest Адрес электронной почты, по которому должно
   * быть отправлено сообщение.
   */
  static MessageBuilder getInstance(String dest) {
    MessageBuilder builder = null;
    ...
    return builder;
  } // getInstance(String)

  /**
   * Установить значение поля заголовка "to".
   */
  abstract void to(String value);

  /**
   * Установить значение поля заголовка "from".
   */
  abstract void from(String value);

  /**
   * Установить значение поля заголовка "organization".
   */
  void organization(String value) { }

  /**
   * Добавить текстовое сообщение.
   */
  abstract void plainText(String content);
```

```
    /**
     * Завершить и возвратить отправляемое сообщение электронной
     * почты.
     */
    abstract OutboundMessageIF getOutboundMsg() ;
} // class MessageBuilder
```

И наконец, так выглядит код для интерфейса `OutboundMessageIF`:

```
public interface OutboundMessageIF {
    public void send() ;
} // interface OutboundMsgIF
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ BUILDER

Interface. Шаблон Builder использует шаблон Interface для сокрытия класса объекта `ProductIF`.

Composite. Объект, созданный при помощи шаблона Builder, обычно является составным.

Factory Method. Шаблон Builder использует шаблон Factory Method для принятия решения, какой класс конкретного строителя инстанциировать.

Template Method. При реализации класса Abstract Builder часто используется шаблон Template Method.

Null Object. Шаблон Null Object может использоваться шаблоном Builder для получения данных из бездействующих реализаций методов.

Visitor. Шаблон Visitor позволяет объекту клиента быть более тесно связанным с построением нового сложного объекта, чем это допускает шаблон Builder. Вместо описания содержимого создаваемых объектов при помощи множественных обращений к методам информация представляется целиком как комплексная структура данных.

Prototype (Прототип)

Этот шаблон был ранее описан в работе [GoF95].

СИНОПСИС

Шаблон Prototype позволяет создавать специальные объекты, ничего точно не зная об их классе или деталях их создания. Механизм можно описать так: объекту, инициирующему создание других объектов, предоставляются объекты-прототипы. Затем этот объект создает объекты путем копирования этих объектов-прототипов.

КОНТЕКСТ

Предположим, что создается программа CAD (Computer-Assisted Design, система автоматизированного проектирования), которая позволит пользователям чертить диаграммы при помощи палитры символов. Программа будет иметь основной набор встроенных символов. Однако ее могут применять для выполнения разных нестандартных задач, для решения которых не подойдет основной набор символов. Поэтому должна существовать возможность предоставления дополнительных наборов символов, которые пользователи смогут при необходимости добавлять в программу.

При этом возникает проблема, каким образом предоставить эти палитры дополнительных символов. Можно без труда организовать программу таким образом, чтобы все символы (и основные, и дополнительные) передавались по наследству от общего родительского класса. При этом остальная часть программы, рисующей диаграммы, получает возможность согласованного управления объектами символов. Но остается открытым вопрос, как программа будет создавать эти объекты. Создание объектов подобного рода зачастую сложнее простого инстанциирования класса. Кроме того, при этом может потребоваться установка значений для атрибутов данных или комбинация объектов с целью создания составного объекта.

Решение состоит в том, чтобы предоставить рисующей программе предварительно созданные объекты с целью использования их в качестве прототипов для создания аналогичных объектов. Самое важное требование для объектов, используемых в качестве прототипов, заключается в том, что они должны иметь метод, обычно с именем `clone`, который возвращает новый объект, являющийся копией исходного объекта (рис. 5.13).

Программа содержит коллекцию объектов-прототипов `Symbol`. Она использует объекты `Symbol`, клонируя их. Объекты `SymbolBuilder` создают объекты `Symbol` и регистрируют их в программе рисования.

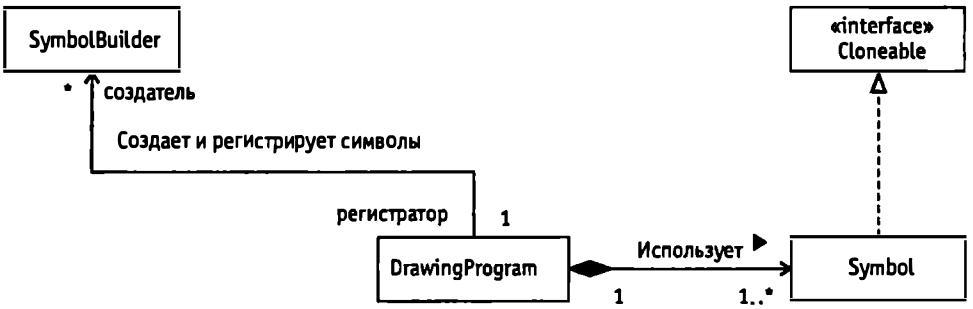


Рис. 5.13. Прототип символа

Все Java-классы наследуют от класса `Object` метод `clone`, который возвращает копию объекта. Клонирование производится только для экземпляров тех классов, которые дают на это разрешение. Класс дает разрешение на клонирование своего экземпляра в том и только в том случае, когда он реализует интерфейс `Cloneable`. Класс `Symbol` реализует интерфейс `Cloneable`, поэтому программа рисования может клонировать объекты `Symbol`, которыми она управляет, и может включать такие объекты в рисунки.

МОТИВЫ

☺ Система должна иметь возможность создавать объекты, ничего точно не зная ни об их классе, ни о том, как они создавались или какие данные содержат.

☺ Инстанциируемые классы не известны системе вплоть до стадии выполнения, когда они запрашиваются «на ходу» некоторыми процессами, например, динамической компоновкой.

☺ Следующие подходы, допускающие создание большого количества разных объектов, являются нежелательными:

1. Классы, инициирующие создание объектов, создают эти объекты напрямую. Это делает такие классы зависимыми от большого количества других классов.

2. Классы, инициирующие создание объектов, создают объекты косвенно, через метод класса-фабрики. Метод класса-фабрики, который способен создавать множество разнообразных объектов, может быть очень большим, и его поддержка может вызывать затруднения.

3. Классы, инициирующие создание объектов, создают объекты косвенно, через абстрактный класс-фабрику. Чтобы абстрактный класс-фабрика мог создавать большое количество различных объектов, он должен иметь множество разнообразных конкретных классов-фабрик, организованных в виде иерархии, параллельной иерархии инстанциируемых классов.

☺ Различные объекты, которые должны создаваться системой, могут быть экземплярами одного и того же класса и отличаться друг от друга информацией, заключенной в них.

РЕШЕНИЕ

Нужно дать возможность классу создавать объекты, которые реализуют известный интерфейс, задавая экземпляр-прототип для создаваемого объекта каждого вида. Затем можно создавать новые объекты, клонируя экземпляр-прототип.

На рис. 5.14 представлена структура шаблона Prototype. Рассмотрим роли, которые играют эти классы и интерфейс.

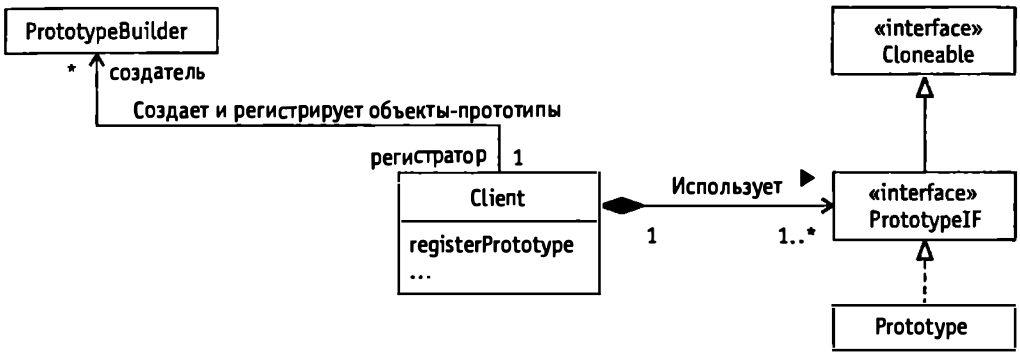


Рис. 5.14. Шаблон Prototype

Client. Класс `Client` представляет часть программы, для которой и предназначен шаблон Prototype. Класс `Client` должен создавать объекты, о которых он знает немного. Он будет содержать метод, который может вызываться с целью добавления объекта-прототипа в коллекцию объектов класса `Client`. На рис. 5.14 этот метод называется `registerPrototype`. Однако для практической реализации больше подходит имя, отражающее вид объекта, представленного прототипом, например, `registerSymbol`.

Prototype. Классы в этой роли реализуют интерфейс `PrototypeIF` и инстанциируются с целью их клонирования клиентом. Классы, выступающие в этой роли, обычно являются абстрактными классами, имеющими определенное количество конкретных подклассов.

PrototypeIF. Все объекты-прототипы должны реализовывать интерфейс, выступающий в этой роли. Класс клиента взаимодействует с объектами-прототипами через этот интерфейс. Интерфейсы, играющие эту роль, должны имплементировать интерфейс `Cloneable`. Таким образом, все объекты, реализующие интерфейс `PrototypeIF`, могут быть клонированы.

PrototypeBuilder. Он соответствует любому классу, который инстанциируется с целью предоставления объектов-прототипов для объекта `Client`. Такие классы

должны иметь имя, обозначающее тип создаваемого ими объекта-прототипа, например, `SymbolBuilder`.

Объект `PrototypeBuilder` создает объекты `Prototype`. Он передает каждый вновь созданный объект `Prototype` методу `registerPrototype` объекта `Client`.

РЕАЛИЗАЦИЯ

Важный вопрос реализации заключается в том, как объекты `PrototypeBuilder` добавляют объекты в принадлежащую клиентским объектам палитру, состоящую из объектов-прототипов. Самая простая стратегия состоит в том, чтобы клиентский класс предоставлял для этой цели метод, который могли бы вызывать объекты `PrototypeBuilder`. Возможным недостатком является то, что объекты `PrototypeBuilder` должны будут знать о классе клиентского объекта. Если это является проблемой, то `PrototypeBuilder` могут быть избавлены от информации о конкретном классе клиентских объектов посредством введения в систему интерфейса или абстрактного класса, реализовывать или наследовать которые должны будут клиентские классы.

Другой важный вопрос заключается в том, как реализовать операцию клонирования объектов-прототипов. Существуют два основных подхода к реализации операции клонирования:

1. *Поверхностное копирование* означает, что переменные клонированного объекта содержат те же значения, что и переменные исходного объекта, и что все ссылки указывают на одинаковые объекты. Другими словами, при поверхностном копировании копируется только клонируемый объект, но не объекты, на которые он ссылается. И оригинал, и поверхностная копия ссылаются на одни и те же объекты.

2. *Глубокое копирование* означает, что переменные клонированного объекта содержат те же самые значения, что и переменные исходного объекта, исключая переменные, которые ссылаются на объекты. Теперь они ссылаются на копии тех объектов, на которые ссылается исходный объект. Другими словами, при глубоком копировании копируется клонируемый объект и те объекты, на которые он ссылается. Глубокая копия ссылается на копии тех объектов, на которые ссылается исходный объект.

Реализация глубокого копирования может быть очень сложной. Нужно будет принимать рещение, делать глубокие или поверхностные копии косвенно копируемых объектов. Кроме того, необходимо очень осторожно обращаться с любыми циклическими ссылками.

Поверхностное копирование реализуется проще, так как все классы наследуют метод `clone` класса `Object`, который легко это делает. Однако если класс объекта не реализует интерфейс `Cloneable`, то метод `clone` не будет работать. Если все объекты-прототипы, используемые программой, будут клонировать сами себя по методу поверхностного копирования, то, объявив интерфейс

`PrototypeIF` как расширение интерфейса `Cloneable`, можно будет сэкономить время. Таким образом, все классы, реализующие интерфейс `PrototypeIF`, будут реализовывать также интерфейс `Cloneable`.

Некоторые объекты, например потоки и сокеты, не могут просто копироваться или совместно использоваться. Какая бы стратегия копирования ни применялась, если имеются ссылки на такие объекты, то для использования скопированных объектов придется создавать эквивалентные объекты.

Если в палитре объекта `Client`, состоящей из объектов-прототипов, количество объектов непостоянно, то неудобно использовать отдельные переменные для ссылки на каждый объект-прототип. Проще использовать объект коллекции, который может содержать динамически расширяющуюся или сужающуюся палитру, состоящую из объектов-прототипов. Объект коллекции, исполняющий эту роль в шаблоне Prototype, называется *управляющим прототипом*. Управляющие прототипами могут быть сложнее, чем простая коллекция. Они могут позволять получать объекты из палитры при помощи значений их атрибутов или других ключей.

Если программа имеет множество клиентских объектов, необходимо рассмотреть другой вопрос. Будут ли клиентские объекты иметь свои собственные палитры, состоящие из объектов-прототипов, или они будут совместно использовать одну и ту же палитру? Ответ на этот вопрос зависит от требований приложения.

СЛЕДСТВИЯ

☺ На стадии выполнения программа может динамически добавлять и удалять объекты-прототипы. Это значительное преимущество, которое не может предложить ни один другой порождающий шаблон проектирования, описанный в данной книге.

☺ Объект `PrototypeBuilder` может просто предоставлять постоянный набор объектов-прототипов.

☺ Объект `PrototypeBuilder` может обеспечивать дополнительную гибкость, допуская создание новых объектов-прототипов посредством комбинации объектов или изменения значений атрибутов объектов.

☺ Клиентский объект тоже может создавать объекты-прототипы новых видов. В примере программы рисования, рассмотренном ранее, клиентский объект мог бы вполне разумно позволить пользователю нарисовать свой значок и затем вставлять его в свою палитру.

☺ Клиентский класс не зависит от конкретного класса объектов-прототипов, который он использует. Кроме того, клиентскому классу не нужно знать подробности создания объектов-прототипов.

☺ Объекты `PrototypeBuilder` инкапсулируют детали построения объектов-прототипов.

☺ Если объекты-прототипы будут реализовывать интерфейс, например, `PrototypeIF`, то шаблон Prototype гарантирует предоставление объектами-прототипами согласованного набора методов, используемого клиентскими объектами.

• Нет необходимости в том, чтобы объекты-прототипы были представлены в виде какой-либо иерархии классов.

☹ Недостатком шаблона Prototype является дополнительное время, затрачиваемое на написание классов `PrototypeBuilder`.

☹ Программы, использующие шаблон Prototype, зависят от динамической компоновки или аналогичных механизмов. Инсталляция таких программ может быть более сложной.

ПРИМЕНЕНИЕ В JAVA API

Шаблон Prototype очень важен для JavaBeans. JavaBeans — это экземпляры классов, которые удовлетворяют определенным соглашениям об именах. Соглашения об именах позволяют программе создания компонентов (Beans) знать, как их настраивать. После настройки объекта компонента с целью использования его в приложении, объект сохраняется в файле, который загружается приложением на стадии выполнения. Этот способ клонирования объектов требует дополнительного времени.

ПРИМЕР КОДА

Предположим, что нужно написать интерактивную ролевую игру, которая позволит пользователю взаимодействовать с персонажами, которые моделируются компьютером. Может быть, игрокам со временем надоест общаться с одними и теми же действующими лицами, и они захотят поиграть с новыми героями. Поэтому необходимо разработать также расширение игры, которое содержит несколько предварительно созданных действующих лиц и программу для создания дополнительных героев.

Используемые в игре действующие лица — это экземпляры относительно небольшого количества классов, например, `Hero`, `Fool`, `Villian` и `Monster`. Различие между всеми этими экземплярами одного и того же класса заключается в различии задаваемых для них значений атрибутов, например, изображений, которые используются для отображения на экране их роста, веса, ума и ловкости.

На рис. 5.15 показаны некоторые классы, используемые в игре.

Ниже представлен код для интерфейса `CharacterIF`. Этот интерфейс выступает в роли `PrototypeIF`.

```
public interface CharacterIF extends Cloneable {
   public String getName() ;
```

```
    public void setName(String name) ;
    public Image getImage() ;
    public void setImage(Image image) ;
    public int getStrength() ;
    public void setStrength(int strength) ;
    ...
} // class CharacterIF
```

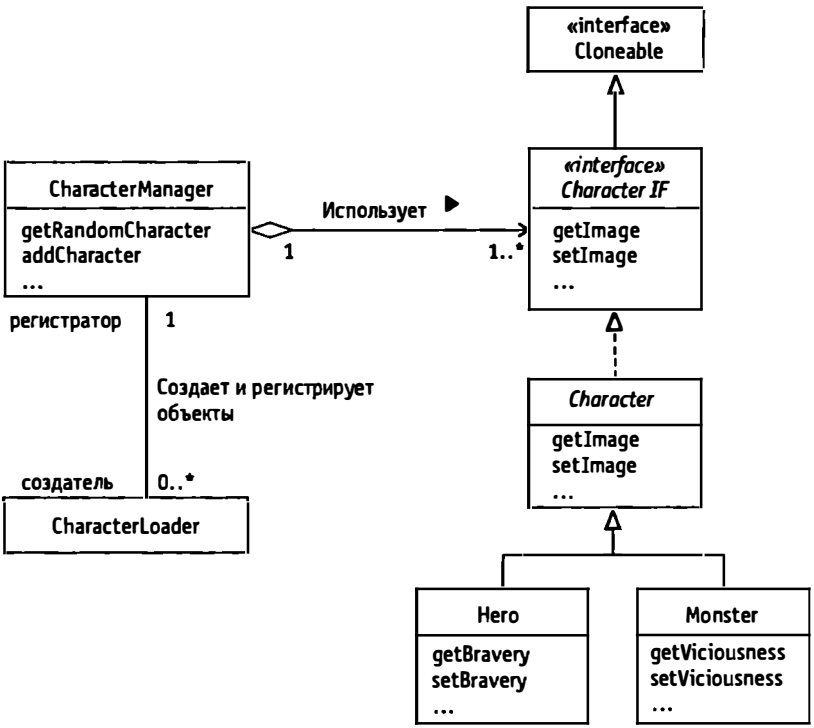


Рис. 5.15. Использование шаблона Prototype

А теперь приведем исходный текст для класса `Character` — абстрактного класса, выступающего в роли `Prototype`:

```
public abstract class Character implements CharacterIF {
...
   /**
    * Замещаем clone - делаем его открытым.
    */
   public Object clone() {
      try {
         return super.clone();
```

```
        } catch (CloneNotSupportedException e) {
            // Этого не должно случиться никогда,
            // так как этот класс реализует Cloneable.
            throw new InternalError();
        } // try
    } // clone()

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public Image getImage() { return image; }

    public void set Image(Image image) { this.image = image; }
...
} // class Character
```

Здесь в основном используются простые методы доступа. Один менее очевидный — метод `clone`. Все объекты наследуют метод клонирования от класса `Object`. Поскольку этот метод не является открытым в классе `Object`, класс `Character` должен замещать его, объявляя открытым и тем самым делая его доступным для других классов.

Приведем исходный текст программы для класса `Hero`. Это один из классов, который выступает в роли прототипа:

```
public class Hero extends Character {
    private int bravery;
...
    public int getBravery() { return bravery; }

    public void setBravery(int bravery) {
        this.bravery = bravery;
    }
} // class Hero
```

Класс `Monster` похож на класс `Hero`.

Приведем код для класса `CharacterManager`, играющего роль клиентского класса:

```
public class CharacterManager {
    private Vector characters = new Vector();
```

```
    /**
     * Возвращаем копию случайно выбранного объекта
     * персонажа из коллекции.
     */
    Character getRandomCharacter() {
      int i = (int)(characters.size()*Math.random());
      Character c = (Character)characters.elementAt(i);
      return (Character)c.clone();
    } // getRandomCharacter()

    /**
     * Добавляем объект-прототип в коллекцию.
     */
    void addcharacter(Character character) {
      characters.addElement(character);
    } // addCharacter(Character)
...
} // class CharacterManager
```

Приведем код для класса `CharacterLoader`, играющего роль `PrototypeBuilder`:

```
    /**
     * Этот класс загружает объекты действующих лиц
     * и добавляет их в CharacterManager.
     */
    class CharacterLoader {
      private CharacterManager mgr;
      /**
        * Constructor
        * @param cm
        *  CharacterManager, с которым будет работать этот объект.
      */
      CharacterLoader(CharacterManager cm) {
        mgr = cm;
      } // Constructor(CharacterManager)

      /**
      * Загружаем объекты действующих лиц из заданного файла.
      * Так как сбой при загрузке влияет только на остальную часть
      * программы, не позволяя добавить в игру новые объекты
      * персонажей, нам не нужно генерировать какие-либо
      * исключения.
      */
```

```
    int loadCharacters(String fname) {
       int objectCount = 0;  // Количество загруженных объектов.
            // Если создание InputStream заканчивается неудачно,
            // просто выходим из метода.
       try {
            InputStream in;
            in = new FileInputStream(fname);
            in = new BufferedInputStream(in);
            ObjectInputStream oIn = new ObjectInputStream(in);
            while(true) {
               Object c = oIn.readObject();
               if (c instanceof Character) {
                  mgr.addCharacter((Character)c);
               } // if
            } // while
         } catch (Exception e) {
         } // try
       return objectCount;
    } // loadCharacters(String)
} // class CharacterLoader
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ PROTOTYPE

Composite. Шаблон Prototype часто используется вместе с шаблоном Composite. Композиция применяется для организации объектов-прототипов.

Abstract Factory. Шаблон Abstract Factory может быть хорошей альтернативой шаблону Prototype в тех случаях, когда не нужны допускаемые шаблоном Prototype динамические изменения палитры объектов-прототипов.

Классы `PrototypeBuilder` могут использовать шаблон Abstract Factory для создания набора объектов-прототипов.

Facade. Клиентский класс обычно действует как класс-фасад, отделяющий другие классы, участвующие в шаблоне Prototype, от остальной части программы.

Factory Method. Шаблон Factory Method может быть альтернативой шаблону Prototype в том случае, если палитра, состоящая из объектов-прототипов, всегда содержит не более одного объекта.

Decorator. Шаблон Prototype часто используется вместе с шаблоном Decorator для составления объектов-прототипов.

Singleton (Одиночка)

Этот шаблон был ранее описан в работе [GoF95].

СИНОПСИС

Шаблон Singleton гарантирует, что создается только один экземпляр некоторого класса. Все объекты, использующие экземпляр этого класса, имеют дело с одним и тем же экземпляром.

КОНТЕКСТ

Некоторые классы должны иметь только один экземпляр. Эти классы обычно имеют дело с централизованным управлением каким-то ресурсом. Ресурс может быть внешним (как в случае с объектом, управляющим многократным использованием соединений баз данных) и внутренним (например, объект, содержащий счетчик ошибок и другие статистические данные, предназначенные для компилятора).

Предположим, что нужно написать класс, который может использоваться аплетом с целью воспроизведения только одного аудиоклипа в какой-то момент. Если апплет содержит два фрагмента кода, которые независимо друг от друга могут воспроизводить аудиоклипы, то существует вероятность, что оба клипа будут воспроизводиться одновременно. В итоге может получиться что угодно, начиная от ситуации, когда пользователи слышат оба аудиоклипа вместе, заканчивая тем, что звуковоспроизводящий механизм платформы не может справиться с одновременным воспроизведением двух аудиоклипов.

Чтобы предотвратить нежелательную ситуацию проигрывания двух аудиоклипов в одно и то же время, создаваемый класс должен прервать проигрывание аудиоклипа до того, как начнет воспроизводиться следующий. Способ проектирования класса для осуществления такой политики, в то же время сохраняя этот класс простым, заключается в том, что должно гарантироваться наличие только одного экземпляра такого класса, совместно используемого всеми объектами, применяющими этот класс. Если все запросы на проигрывание аудиоклипов идут через один и тот же объект, то этот объект может просто прервать какой-то аудиоклип перед началом воспроизведения следующего. На рис. 5.16 представлен такой класс.

Конструктор класса `AudioClipManager` является закрытым. Это запрещает другому классу прямым образом создавать экземпляр класса `AudioClipManager`. Вместо этого для получения экземпляра класса `AudioClipManager` другие классы должны вызывать его метод `getInstance`. Это статический метод, который всегда возвращает один и тот же экземпляр класса `AudioClipManager`.

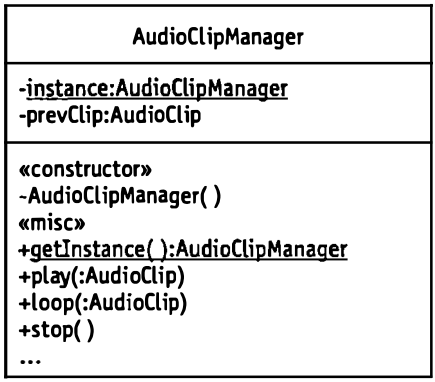


Рис. 5.16. Класс, управляющий аудиоклипом

Возвращаемый им экземпляр — это тот экземпляр, на который ссылается его закрытая статическая переменная `instance`.

Остальные методы класса `AudioClipManager` отвечают за контроль над воспроизведением аудиоклипов. Класс `AudioClipManager` имеет закрытую постоянную переменную экземпляра `prevClip`, которая вначале установлена в `null`, а позднее указывает на последний проигрывавшийся аудиоклип. Перед воспроизведением нового аудиоклипа экземпляр класса `AudioClipManager` останавливает аудиоклип, на который ссылается `prevClip`. Это гарантирует, что предыдущий затребованный аудиоклип останавливается перед началом проигрывания следующего.

МОТИВЫ

☺ Должен существовать по крайней мере один экземпляр некоторого класса. Даже если методы класса не используют данных экземпляра или используют только статические данные, может понадобиться экземпляр такого класса по разным причинам. Некоторые самые общие причины могут заключаться в том, что нужен экземпляр для передачи параметра методу другого класса или необходимо иметь косвенный доступ к классу, через интерфейс.

☺ Не должно быть более одного экземпляра класса. Это может объясняться тем, что нужно иметь только один источник некоторой информации. Например, чтобы был единственный объект, который отвечает за генерирование последовательности порядковых номеров.

☺ Один экземпляр класса должен быть доступен для всех клиентов этого класса.

☹ Создание объекта не требует больших затрат, но он занимает большой объем памяти или непрерывно на протяжении всего своего времени жизни использует другие ресурсы.

РЕШЕНИЕ

Шаблон Singleton достаточно прост, поскольку он содержит только один класс (рис. 5.17).

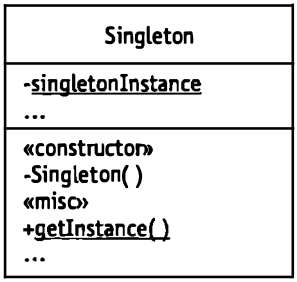


Рис. 5.17. Класс-одиночка

Класс-одиночка имеет статическую переменную, ссылающуюся на такой экземпляр класса, который нужно использовать. Этот экземпляр создается при загрузке класса в память. Необходимо реализовать класс таким образом, чтобы запретить другим классам создавать любые дополнительные экземпляры класса-одиночки. Это значит, что необходимо сделать все конструкторы класса закрытыми.

Для доступа к одному экземпляру класса-одиночки класс предоставляет статический метод, обычно с именем `getInstance` или `getClassname`, который возвращает ссылку на единственный экземпляр класса.

РЕАЛИЗАЦИЯ

Хотя шаблон Singleton предполагает относительное простое решение, его реализация включает очень большое количество тонких моментов.

Закрытый конструктор

Чтобы подчеркнуть природу класса-одиночки, необходимо написать код для этого класса таким образом, чтобы запретить другим классам прямым образом создавать экземпляры этого класса. Поэтому все конструкторы класса должны быть объявлены закрытыми. В данном случае обязательно нужно объявить хотя бы один закрытый конструктор. Если класс вообще не объявляет никаких конструкторов, то Java автоматически создаст для него открытый конструктор по умолчанию.

«Ленивое» инстанциирование

Широко распространенный вариант шаблона `Singleton` используется в ситуациях, при которых экземпляр класса-одиночки может не понадобиться. Неиз-

вестно, будет ли программа нуждаться в использовании экземпляра одиночки до тех пор, пока не будет в первый раз вызван метод `getInstance`. В подобных ситуациях откладывают создание экземпляра класса до момента первого обращения к методу `getInstance`. Технология задержки инстанциирования объекта до тех пор, пока оно действительно не понадобится, иногда называется «ленивым» (lazy) инстанциированием.

Разрешение на создание более одного экземпляра

Существует другая версия шаблона Singleton. Поскольку политика инстанциирования класса инкапсулирована в его методе `getInstance`, то существует возможность изменять методику создания класса. Один из способов заключается в том, чтобы заставить метод `getInstance` в качестве альтернативы возвращать один из двух возможных экземпляров или периодически создавать новый экземпляр, который будет возвращаться методом `getInstance`.

Ситуация такого рода может возникнуть в том случае, когда используются небольшие постоянные отдельные объекты для представления внешних ресурсов и нужно сбалансировать их загрузку. Обобщая такую ситуацию и распространяя ее на произвольное или переменное количество объектов, получают шаблон Object Pool.

Создание копий объекта-одиночки

Объект-одиночка обычно должен быть единственным экземпляром своего класса. Даже в тех случаях, когда допускается существование более чем одного экземпляра объекта-одиночки, обычно нужно, чтобы создание объектов-одиночек находилось под полным контролем со стороны метода `getInstance` класса-одиночки. Это означает, что нельзя, чтобы какие-либо другие классы копировали объект-одиночку.

Из вышесказанного следует, что класс-одиночка не должен реализовывать интерфейс `java.lang.Cloneable`. Другой класс не должен иметь возможность делать копирование, вызывая метод `clone` объекта-одиночки.

Сериализация — это механизм, который предоставляет Java для преобразования содержимого объектов в байтовый поток[1]. Десериализация — это механизм, предоставляемый Java для преобразования потока байтов (созданного при сериализации) в объекты, имеющие такое же содержимое, которое имели первоначальные объекты. Полнее сериализация и десериализация рассматриваются при описании шаблона Snapshot (см. гл. 8).

---

[1] Подробное описание сериализации можно найти на Web-странице фирмы Sun, посвященной языку Java: java.sun.com. В момент написания данной книги URL-адрес, содержащий описание сериализации, выглядит так: http://java.sun.com/j2se/1.3/docs/guide/serialization/.

Сериализация может быть использована для копирования объектов, но, как правило, копирование не подразумевает применение сериализации. Сериализация предназначена для использования в двух следующих общих случаях.

1. Сериализация применяется для того, чтобы сделать объекты постоянными. Это достигается посредством преобразования объекта в поток байтов и записи в файл и позволяет в будущем восстановить объекты. Именно такой механизм используется при сохранении и восстановлении Java Beans.
2. Сериализация используется для поддержки удаленного вызова процедуры, использующей RMI (Remote Method Invocation, удаленный вызов метода). RMI используется EJB (Enterprise Java Beans, серверные компоненты Java), и RMI использует сериализацию для передачи значений аргументов, а также для передачи возвращаемого значения назад вызвавшей стороне.

Обычно эти функции и не выполняются над объектом-одиночкой. В целом объекты `Singleton` не должны участвовать в процессе сериализации. Этого можно добиться, если никогда прямо не сериализировать объект-одиночку и не иметь никаких классов, хранящих ссылку на объект-одиночку в своих переменных экземпляров. Вместо этого следует заставить клиентов класса-одиночки вызывать методы `getInstance` этого класса каждый раз, когда они хотят получить доступ к объекту-одиночке.

Конкурентные обращения к методу getInstance

Если существует хоть какая-нибудь вероятность того, что многочисленные потоки могут вызывать метод `getInstance` класса-одиночки одновременно, то необходимо убедиться в том, что метод `getInstance` не создает многочисленных экземпляров класса-одиночки. Рассмотрим следующий код:

```
public class Foo {
   private Foo myInstance;
   ...
   public static Foo getInstance() {
      if (myInstance==null) {
         myInstance = new Foo();
      } // if
      return myInstance;
   } // getInstance()
   ...
} // class Foo
```

Если два потока вызывают метод `getInstance` в одно и то же время и не было никаких предыдущих обращений к этому методу, то оба вызова увидят, что значение `myInstance` равно `null`, и оба вызова создадут экземпляр класса `Foo`.

Чтобы предотвратить возникновение этой проблемы, можно объявить метод `getInstance` как синхронизированный. Тогда в любой момент времени только один поток имеет возможность выполнить метод `getInstance`.

При объявлении метода `getInstance` синхронизированным увеличивается время его выполнения, поскольку при каждом обращении к этому методу перед его выполнением должна выполняться блокировка.

Неявная ошибка

Существует вероятность появления трудно обнаруживаемой ошибки в разных вариантах реализации шаблона Singleton. Эта ошибка может заставить класс-одиночку создавать и инициализировать несколько своих экземпляров. Проблема возникает в программах, которые ссылаются на класс-одиночку только через другие динамически загружаемые классы, описанные в шаблоне Dynamic Linkage (см. гл. 7).

Некоторые программы организованы так, что они динамически загружают набор классов, используют эти классы в течение какого-то времени, а затем прекращают их использовать. Апплеты, сервлеты и мидлеты[1] управляются именно таким образом. Если программа перестает использовать классы, они могут быть удалены сборщиком мусора. Все это замечательно. Если программа не поддерживает ссылки на классы после того, как она перестала с ними работать, и для классов разрешена сборка мусора, то неиспользуемые классы будут в конечном счете удалены сборщиком мусора.

Такое поведение может представлять проблему для класса-одиночки. Когда класс-одиночка удаляется при сборке мусора, он будет загружаться снова, если существует другая динамическая ссылка на него. После загрузки класса во второй раз очередной запрос на получение его экземпляра будет возвращать новый экземпляр. Это может привести к неожиданным результатам.

Предположим, есть класс-одиночка, в задачу которого входит поддержка статистических данных, касающихся рабочих показателей. Посмотрим, что случится, если такой класс будет удален при сборке мусора и загружен повторно. Первый раз его метод `getInstance` вызывается после того, как класс был загружен второй раз, и он создает новый объект. Его переменные экземпляра будут иметь свои начальные значения, и ранее собранные статистические данные будут утеряны.

Если класс загружается объектом `ClassLoader`, то он не будет удален сборщиком мусора до тех пор, пока для объекта `ClassLoader` не будет разрешена сборка мусора. Если нельзя на практике осуществить такое управление временем жизни объекта-одиночки, то существует более общий способ. Он заключается в том, чтобы обеспечить существование ссылки, прямой или косвенной, от

[1] Мидлеты — от англ. MIDlets (MID — Mobile Information Devices, мобильные информационные устройства). (*Примеч. ред.*)

вующего потока на объект, который не должен удаляться сборщиком му-
Приведенный здесь класс может быть использован только для этого:

```
public class ObjectPreserver implements Runnable {
   // Это защищает данный класс и все, на что он ссылается,
   // от удаления при сборке мусора.
   private static ObjectPreserver lifeLine
      = new ObjectPreserver();

   // Этот класс не должен удаляться при сборке мусора, поэтому
   // не будет удален ни этот HashSet,
   // ни объект, на который он ссылается.
   private static HashSet protectedSet = new HashSet();

   private ObjectPreserver() {
      new Thread(this).start();
   } // constructor()

   public synchronized void run() {
      try {
         wait();
      } catch (InterruptedException e) {
      } // try
   } // run()

   /**
    * Собранная сборщиком мусора и переданная этому методу
    * коллекция объектов будет сохранена до тех пор,
    * пока объекты не будут переданы методу unpreserveObject.
    */
   public static void preserveObject(Object o) {
      protectedSet.add(o);
   } // preserveObject()

   /**
    * Объекты, переданные этому методу, теряют защиту от сборщика
    * мусора.
    */
   public static void unpreserveObject(Object o) {
      protectedSet.remove(o);
   } // unpreserveObject(Object)
} // class ObjectPreserver
```

ии объект класса, инкапсулирующий класс или один из экземпляров класса, ледается методу `preserveObject` класса `ObjectPreserver`, представленному оследнем листинге, то этот класс не будет удален при сборке мусора.

ІЕДСТВИЯ

Существует строго один экземпляр класса-одиночки.

Метод `getInstance` класса-одиночки инкапсулирует политику создания для класса-одиночки, поэтому классы, использующие класс-одиночку, не зависят от деталей его инстанциирования.

Другие классы, которые хотят ссылаться на один экземпляр класса-одиночки, должны получать этот экземпляр, вызывая статический метод `getInstance` класса, а не создавать экземпляр самостоятельно.

Образование подклассов от класса-одиночки вызывает затруднения и приводит к неправильно инкапсулированным классам. Чтобы создать подкласс класса-одиночки, необходимо объявить в нем конструктор, который не будет закрытым. Кроме того, поскольку статические функции не могут быть замещены, подкласс класса-одиночки должен оставлять метод `getInstance` своего суперкласса открытым.

ИМЕНЕНИЕ В JAVA API

:с Java API с именем `java.lang.Runtime` — это класс-одиночка. Он имеет го один экземпляр. У него нет открытых конструкторов. Чтобы получить ку на единственный его экземпляр, другие классы должны вызывать его ический метод `getRuntime`.

ІМЕР КОДА

е показан реализованный на языке Java класс, который можно использоля запрета одновременного воспроизведения двух аудиоклипов. Класс яв:я классом-одиночкой. Можно получить доступ к его экземпляру, вызывая гатический метод `getInstance`. Если вы проигрываете аудиоклип при по- такого объекта, он прерывает воспроизведение последнего аудиоклипа началом воспроизведения нового аудиоклипа. Если все аудиоклипы восзводятся через объект `AudioClipManager`, то в любой момент времени гда не будет воспроизводиться более одного аудиоклипа.

```
public class AudioClipManager implements AudioClip{
   private static AudioClipManager instance
      = new AudioClipManager();
   private AudioClip prevClip; // Предыдущий аудиоклип.
```

```
    /**
     * Объявляем закрытый конструктор. Таким образом,
     * компилятор не будет создавать открытый конструктор
     * по умолчанию.
     */
    private AudioClipManager() { }

    /**
     * Возвращает ссылку на единственный экземпляр этого класса.
     */
    public static AudioClipManager getInstance() {
      return instance;
    } // getInstance()
...
    /**
     * Останавливаем предыдущий аудиоклип
     * и воспроизводим заданный аудиоклип.
     * @param clip Новый аудиоклип, который должен проигрываться.
     */
    public void play(AudioClip clip) {
      if (prevClip != null)
        prevClip.stop();
      prevClip = clip;
      clip.play();
    } // play(AudioClip)
...
    /**
     * Останавливаем предыдущий аудиоклип
     * и воспроизводим данный аудиоклип в цикле.
     * @param clip Новый аудиоклип, который должен проигрываться.
     */
    public void loop(AudioClip clip) {
      if (prevClip != null)
        prevClip.stop();
      prevClip = clip;
      clip.loop();
    } // play(AudioClip)

    /**
     * Останавливаем воспроизведение этого аудиоклипа.
     */
```

```
    public void stop() {
      if (prevClip != null)
        prevClip.stop();
    } // stop()
} // class AudioClipManager
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ SINGLETON

Шаблон Singleton можно использовать со многими другими шаблонами. В частности, он часто применяется вместе с шаблонами Abstract Factory, Builder и Prototype.

Cache Management. Шаблон Singleton имеет некоторое сходство с шаблоном проектирования Cache Management. С точки зрения функциональности шаблон Singleton подобен шаблону Cache, содержащему только один объект.

Object Pool. Шаблон Object Pool предназначен для управления любой большой коллекцией похожих объектов, а не только одним объектом.

Object Pool (Пул объектов)

СИНОПСИС

Управляет повторным использованием объектов в случаях, когда создание объекта требует больших затрат или может быть создано только ограниченное количество объектов некоторого вида.

КОНТЕКСТ

Предположим, нужно написать библиотеку классов для предоставления доступа к базе данных. Клиенты отправляют запросы к базе данных через сетевое соединение. Сервер базы данных получает и возвращает запросы через то же сетевое соединение.

Чтобы программа могла отправлять запросы базе данных, она должна иметь с ней соединение. Для программистов, которые будут пользоваться библиотекой, самый подходящий способ управления соединениями состоит в том, чтобы каждая часть программы, нуждающаяся в соединении, создавала собственное соединение. Однако создание излишних соединений с базой данных, которые не потребуются на практике, нежелательно по нескольким причинам:

- создание каждого соединения с базой данных может потребовать некоторого времени;
- чем больше соединений, тем больше требуется времени для создания новых соединений;
- каждое соединение с базой данных использует сетевое соединение. Некоторые платформы накладывают ограничения на количество разрешенных ими сетевых соединений.

Проект библиотеки должен примирить следующие конфликтующие стороны. Необходимость обеспечить удобный API для программистов «тянет» проект в одну сторону. Большие затраты на создание объектов соединений с базой данных и, кроме того, возможное ограничение на количество одновременных соединений «тянут» проект в другую сторону. Разрешить это противоречие возможно в том случае, если библиотека будет управлять соединениями с базой данных.

Стратегическое решение, используемое библиотекой для управления соединениями, основано на предположении, что соединения с базой данных программы являются взаимозаменяемыми. Пока соединение находится в состоянии, позволяющем ему передавать запросы к базе данных, неважно, какое из соеди-

нений базы данных программы используется. Учитывая это замечание, библиотека для доступа к базе данных проектируется так, чтобы осуществлялась двухуровневая реализация соединений с базой данных.

Класс `Connection` реализует верхний уровень. Программы, которые используют библиотеку доступа к базе данных, прямым образом создают и используют объекты `Connection`. Объекты `Connection` идентифицируют базу данных, но в них непрямым образом инкапсулировано соединение с этой базой данных. Объект `Connection` образует пару с объектом `ConnectionImpl` только в течение того времени, когда он используется для передачи запроса к базе данных и получения результата. Объекты `ConnectionImpl` инкапсулируют реальное соединение с базой данных.

Библиотека создает и управляет объектами `ConnectionImpl`. Управление объектами `ConnectionImpl` осуществляется путем поддержки пула для тех объектов, которые не образуют в данный момент пару с объектом `Connection`. Библиотека создает объект `ConnectionImpl` только тогда, когда должна образоваться еще одна пара этого объекта с объектом `Connection`, а пул объектов `ConnectionImpl` пуст. На диаграмме классов, представленной на рис. 5.18, изображены классы, участвующие в управлении пулом объектов `ConnectionImpl`.

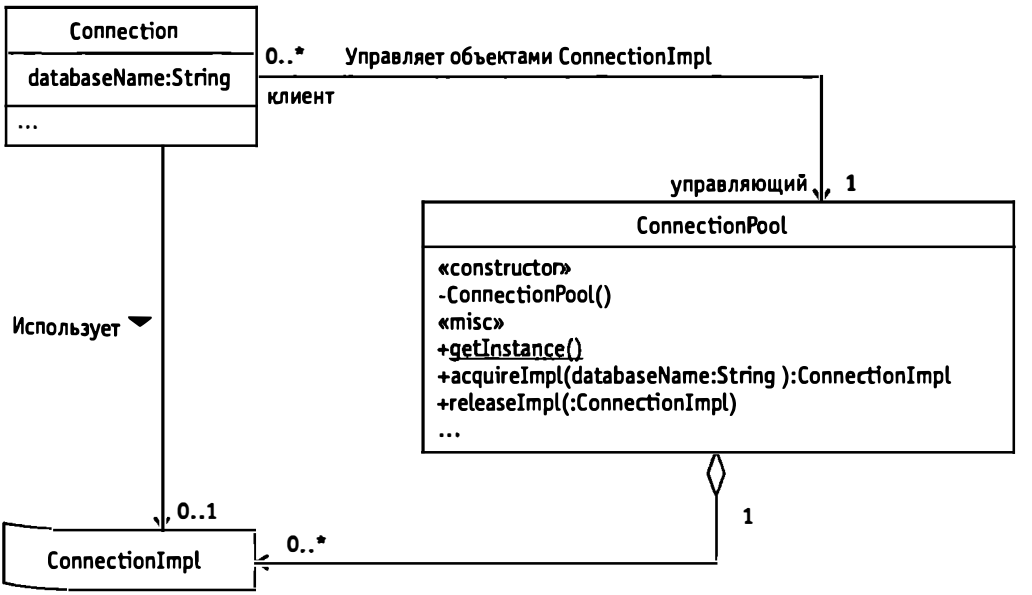


Рис. 5.18. Управление пулом объектов ConnectionImpl

Если объекту `Connection` нужен объект `ConnectionImpl`, он вызывает метод `AcquireImpl` объекта `ConnectionPool`, передавая ему имя базы данных, с которой должно быть установлено соединение. Если в коллекции объектов `ConnectionPool` есть какие-либо объекты `ConnectionImpl`, связанные с нуж-

ной базой данных, то он возвращает один из этих объектов. Если в коллекции объектов `ConnectionPool` нет таких объектов `ConnectionImpl`, он пытается создать такой объект и возвратить его. В случае невозможности создания такого объекта `ConnectionImpl`, он ожидает до тех пор, пока существующий объект `ConnectionImpl` не возвратится в пул посредством вызова метода `releaseImpl`, а затем он возвращает этот объект.

Класс `ConnectionPool` — это класс-одиночка. Должен существовать только один экземпляр класса `ConnectionPool`. Конструктор класса является закрытым. Другие классы обращаются к единственному экземпляру класса `ConnectionPool`, вызывая его статический метод `getInstance`.

Существует множество причин, не позволяющих методу `AcquireImpl` объекта `ConnectionPool` создавать объект `ConnectionImpl`. Одной из таких причин может быть существование ограничения на количество объектов `ConnectionImpl`, создаваемых для подключения к одной и той же базе данных. Причиной этого ограничения является гарантия того, что база данных может поддерживать некоторое минимальное количество клиентов. Существует максимальное количество соединений, которое может поддерживать каждая база данных, поэтому ограничение количества соединений каждого клиента с базой данных позволяет гарантировать поддержку некоторого минимального количества клиентских программ.

МОТИВЫ

- ☺ Программа может создавать только ограниченное количество экземпляров некоторого класса.
- ☺ Создание экземпляров некоторого класса требует довольно больших затрат, поэтому следует избегать создания новых экземпляров такого класса.
- ☺ Программа может избежать создания новых объектов, если будет повторно использовать объекты, с которыми она уже завершила работу, и не позволять сборщику мусора удалять их.
- ☺ Экземпляры класса являются взаимозаменяемыми. Если есть сразу несколько экземпляров, можно выбрать любой из них для использования в своих целях.
- ☺ Ресурсами можно управлять централизованно, с помощью одного объекта, или децентрализованно, с помощью нескольких объектов. Проще управлять ресурсами централизованно, с помощью единственного объекта.
- ☺ Некоторые объекты потребляют такие ресурсы, запасы которых ограничены. Другие объекты могут потреблять много памяти. Третьи объекты могут периодически проверять правильность выполнения некоторого условия, используя ресурсы компьютера и загружая сеть. Если ресурсы, потребляемые объектом, ограничены, то немаловажно, чтобы объект прекратил использовать такой ресурс, когда сам объект не используется.

РЕШЕНИЕ

Если экземпляры некоторого класса можно многократно использовать, избегайте создания новых экземпляров этого класса и используйте их повторно. На рис. 5.19 представлена диаграмма классов шаблона Object Pool.

Опишем роли, исполняемые классами в шаблоне Object Pool.

Reusable. Экземпляры классов в этой роли взаимодействуют с другими объектами в течение ограниченного времени, а затем они больше не нужны для этого взаимодействия.

Client. Экземпляры классов в этой роли используют объекты `Reusable`.

ReusablePool. Экземпляры классов в этой роли управляют объектами `Reusable`, предназначенными для использования объектами `Client`. Как правило, желательно поддерживать в одном и том же пуле все объекты `Reusable`, которые не используются в данный момент, чтобы ими можно было управлять в соответствии с единой последовательной политикой. Для достижения этой цели класс `ReusablePool` проектируется как класс-одиночка. Его конструктор(ы) является закрытым, что вынуждает другие классы обращаться к его методу `getInstance` за получением одного экземпляра класса `ReusablePool`.

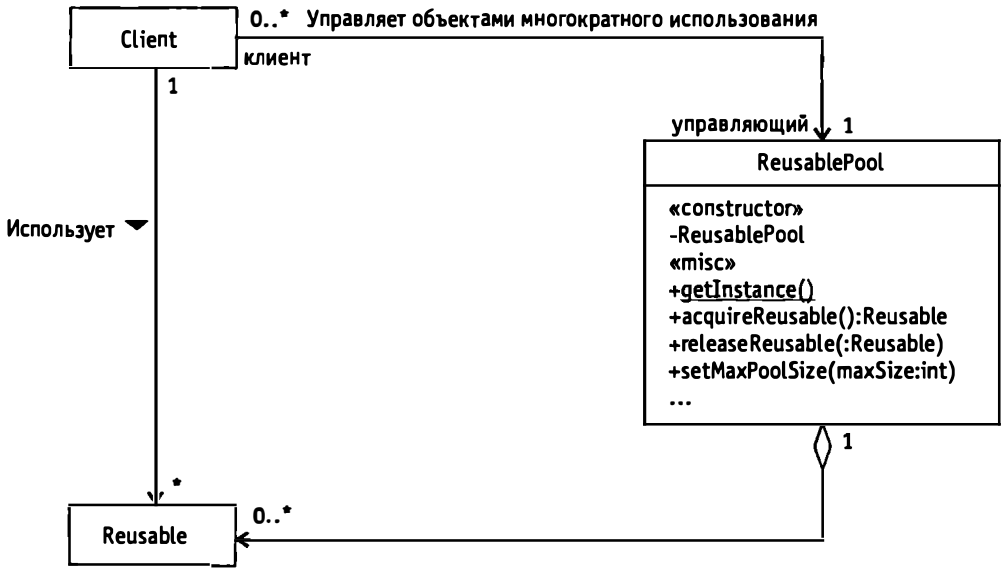


Рис. 5.19. Шаблон Object Pool

Если объекту `Client` потребуется объект `Reusable`, то он вызовет метод `acquireReusable` объекта `ReusablePool`. Объект `ReusablePool` поддерживает коллекцию объектов `Reusable` и применяет ее для поддержки пула объектов `Reusable`, не использующихся в данный момент. Если при вызове метода

cquireReusable в пуле имеются какие-то объекты Reusable, то он удаляет бъект Reusable из пула и возвращает его. Если пул пустой, то метод cquireReusable, если может, создает объект Reusable. Если метод cquireReusable не может создать новый объект Reusable, то он ожидает, ока объект Reusable не будет возвращен в коллекцию.

осле завершения работы с объектом объекты Client передают объект Resable етоду releaseReusable объекта ReusablePool. Метод releaseReusable воз- ращает объект Reusable в пул неиспользующихся объектов Reusable.

о многих приложениях, в которых используется шаблон Object Pool, огрании- ивается общее количество создаваемых объектов Reusable. В таких случаях бъект ReusablePool, создающий объекты Reusable, отвечает за то, чтобы не оздавать такое количество объектов Reusable, которое бы превышало задан- ый максимум. Если объекты ReusablePool отвечают за ввод ограничений ко- ичества создаваемых ими объектов, то класс ReusablePool будет содержать етод, задающий максимальное количество создаваемых объектов. На рис. 5.19 тот метод указан под именем setMaxPoolSize.

РЕАЛИЗАЦИЯ

ри рассмотрении задачи реализации шаблона Object Pool следует рассмотреть екоторые вопросы.

Ограничение максимального количества

Считается, что во многих случаях объект, управляющий пулом объектов, дол- жен ограничивать количество создаваемых экземпляров некоторого класса. Эта проблема решается просто. Но для того, чтобы соблюдать это ограничение оличества создаваемых объектов, объект, отвечающий за управление пулом бъектов, должен быть единственным объектом, способным создавать эти бъекты.

Можно сделать так, что класс будет инстанциироваться только с помощью того ласса, который управляет пулом объектов. Для этого можно задать конструк- ор(ы) управляемого класса закрытым и реализовать класс, управляющий пу- ом, как статический класс-член управляемого класса. Если не контролируется труктура класса, экземплярами которого нужно управлять, можно применить ышеописанный подход, используя наследование.

Структура данных

Если существует ограничение, накладываемое на количество создаваемых объ- ктов или на предельный размер пула объектов, то обычно наилучший способ еализации пула объектов состоит в использовании простого массива. Если на азмер пула объектов не накладывается ограничений, то подходящий способ еализации пула объектов заключается в использовании ArrayList.

Использование «мягких» ссылок (soft references)

Шаблон Object Pool хранит незадействованные объекты. Если программе, применяющей пул объектов, не хватает памяти, то желательно, чтобы сборщик мусора мог удалить объекты из пула и высвободить занимаемую ими память. Чтобы реализовать эту идею, можно использовать «мягкие» ссылки.

«Мягкие» ссылки реализованы в Java API классом `java.lang.ref.SoftReference`. Ссылка на другой объект передается конструктору объекта `SoftReference`. Сразу после создания объекта `SoftReference` его метод `get` возвращает ссылку на объект, который был передан его конструктору. Самое интересное в объектах `SoftReference` то, что они специально предназначены для сборщика мусора.

Если пул объектов ссылается на объекты в пуле, используя «мягкие» ссылки, то программа сборки мусора освободит память, занимаемую объектами, если на эти объекты нет других ссылок и JVM (Java Virtual Machine, виртуальная машина Java) почти исчерпала всю память.

Ограничение размера пула

Существуют такие ситуации, когда «мягкие» ссылки нельзя назвать хорошим решением проблемы ограничения избыточных ресурсов, используемых объектами, которыми управляет пул:

- если управляемые объекты используют внешний ресурс, иногда можно достаточно долго ждать, пока программа сборки мусора удалит объект перед высвобождением ресурса;
- если программе нужна JVM из старых версий (старше версии 1.2), нельзя использовать «мягкие» ссылки.

В качестве альтернативы использования «мягких» ссылок можно применять ограничение, накладываемое на количество объектов, находящихся в пуле. Если задан такой предел количества объектов, он, как правило, меньше того количества объектов, которым пул разрешает существовать одновременно.

Если управляемый пулом объект больше не нужен, он высвобождается, и `ReusablePool` добавляет его к себе в коллекцию. Если пул уже содержит максимальное количество объектов, то высвобожденный объект не добавляется в пул. Если этот объект связан с каким-либо внешним ресурсом, он должен освободить его. Поскольку объект не добавляется в пул, теперь его вполне может удалить сборщик мусора.

Способы ограничения размеров пула и использования «мягких» ссылок, как правило, не применяются одновременно. Программа сборки мусора может очистить объект `SoftReference` в любой момент времени, поэтому никто не может гарантировать, что пул всегда имеет правильное значение счетчика количества объектов, содержащихся в этом пуле.

Управление объектами состояний

Одно из предположений, лежащих в основе идеи пула объектов, состоит в том, что управляемые им объекты являются взаимозаменяемыми. Когда клиент запрашивает объект из пула, он предполагает, что ему известно состояние этого объекта. Клиенты могут изменять состояние объектов, управляемых пулом, поэтому пул должен иметь возможность гарантировать, что его объекты возвращаются в ожидаемое состояние. Для решения этой проблемы чаще всего применяются следующие способы.

- Пул явным образом восстанавливает состояние объекта перед тем, как клиент запросит этот объект. В некоторых случаях это сделать невозможно. Например, если однажды соединение с базой данных было закрыто, пул может не открыть его снова. Кроме того, действительно не нужно, чтобы клиенты закрывали соединение с базой данных.
- Можно запретить клиентам изменять состояние объектов, управляемых пулом, используя шаблон Decorator. Например, чтобы сделать это для соединения с базой данных, следует создать объект-обертку (wrapper), который будет реализовать тот же самый интерфейс, что и реальный объект соединения с базой данных. Объект-обертка должен делегировать все свои методы реальному объекту соединения с базой данных, за исключением метода, который закрывает соединение. Такой метод может быть реализован как бездействующий.

СЛЕДСТВИЯ

☺ Использование шаблона Object Pool позволяет избежать необходимости создания объекта. Он наиболее эффективен в тех ситуациях, когда потребность в объектах не сильно изменяется со временем.

☺ Поддерживая логику управления созданием и многократным использованием экземпляров класса в некотором классе, отделенном от остальных классов, экземпляры которых подлежат управлению, получаем в результате более связанный проект. При этом устраняется взаимодействие между реализацией политики создания и повторного использования и реализацией функциональности управляемого класса.

ПРИМЕР КОДА

Реализации шаблона Object Pool используют один из двух способов, который гарантирует, что управляемые объекты не занимают слишком большой объем памяти. Такие способы либо используют «мягкие» ссылки, указывающие на объекты, либо ограничивают количество объектов, которые могут находиться в пуле. Поскольку эти способы отличаются друг от друга, в этом разделе приводится пример применения каждого из них. Оба класса являются универсальными. Они могут использоваться для задания пула, содержащего объекты любого вида.

Ниже приводится исходный текст для класса, который реализует пул объектов, использующий «мягкие» ссылки.

```
public class SoftObjectPool implements ObjectPoolIF {
   /**
    * Эта коллекция содержит управляемые объекты.
    */
   private ArrayList pool ;
```

Поскольку не существует постоянное ограничение количества объектов, которые могут находиться в пуле, ожидая повторного использования, этот класс для содержания своей коллекции объектов применяет объект `ArrayList`.

Этот класс является общим, поэтому он ничего не знает о том, как создаются управляемые им объекты. Вместо этого он делегирует ответственность за создание объектов объекту, который реализует интерфейс `CreationIF`. Листинг интерфейса `CreationIF` находится в конце данного раздела.

```
   private CreationIF creator ;
```

Экземпляры этого класса отвечают за ограничение количества объектов, управляемых пулом и существующих одновременно. В переменной `instanceCount` содержится число, содержащее количество объектов, существующих в данный момент. Переменная `maxInstances` содержит число, отражающее максимальное количество управляемых пулом объектов. Пул объектов не будет создавать новый объект до тех пор, пока значение `instanceCount` меньше значения, содержащегося в `maxInstances`.

```
   private int instanceCount;
   private int maxInstances ;
```

Поскольку этот класс представляет собой универсальный пул объектов, он не знает заранее, какого рода объектами он должен будет управлять по желанию управляемого объекта. С целью простой профилактической проверки этот класс требует, чтобы класс передавался конструктору как один из параметров. Когда объект высвобождается и помещается в пул, пул должен убедиться, что этот объект является экземпляром определенного класса. Класс управляемых объектов хранится в переменной `poolClass`.

```
   private Class poolClass;

   /**
    * Constructor
    *
    * @param poolClass
    *     Инстанциируемый класс, используемый для создания
    *     объектов пула.
```

```
     * @param creator
     *     Объект, которому пул будет делегировать ответственность
     *     за создание управляемых им объектов.
     */
    public SoftObjectPool( Class poolClass,
                           CreationIF creator ) {
      this( poolClass, creator, Integer.MAX_VALUE);
    } // constructor(Class, CreationIF, int)

    /**
     * Constructor
     *
     * @param poolClass
     *     Инстанциируемый класс, используемый для создания
     *     объектов пула.
     * @param creator
     *     Объект, которому пул будет делегировать ответственность
     *     за создание управляемых им объектов.
     * @param maxInstances
     *     Максимальное количество экземпляров класса poolClass,
     *     которым пул разрешает существовать одновременно. Если пулу
     *     поступает запрос на создание экземпляра класса
     *     poolClass, когда в пуле нет объектов, ожидающих
     *     повторного использования, и имеется множество таких
     *     управляемых пулом объектов, которые используются
     *     в данный момент, то пул не будет
     *     создавать объект до тех пор, пока в пул не будет
     *     возвращен объект для повторного использования.
     */
     public SoftObjectPool( Class poolClass,
                            CreationIF creator,
                            int maxInstances ) {
      this.creator = creator;
      this.poolClass = poolClass;
      pool = new ArrayList();
    } // constructor(Class, CreationIF, int, int)

    /**
     * Возвращает количество объектов в пуле, ожидающих повторного
     * использования. Реальное количество может быть меньше
     * этого значения, поскольку возвращаемая
```

```
 * величина — это количество "мягких" ссылок в пуле. Некоторые
 * или все такие "мягкие" ссылки могут быть очищены сборщиком
 * мусора.
 */
public int getSize() {
   synchronized (pool) {
        return pool.size();
   } // synchronized
} // getSize()

/**
 * Возвращает количество управляемых пулом объектов,
 * существующих в данный момент.
 */
public int getInstanceCount() {
   return instanceCount;
} // getInstanceCount()

/**
 * Возвращает максимальное количество управляемых пулом
 * объектов, которым пул разрешает существовать одновременно.
 */
public int getMaxInstances() {
   return maxInstances;
} // getMaxInstances()

/**
 * Задает максимальное количество управляемых пулом объектов,
 * которым этот пул разрешает существовать одновременно.
 *
 * Если к пулу поступает запрос на создание экземпляра класса
 * poolClass, когда в пуле нет объектов, ожидающих
 * повторного использования, и имеется множество таких
 * управляемых пулом объектов, которые используются в данный
 * момент, то пул не будет создавать объект до тех пор, пока
 * в пул не будет возвращен объект для повторного
 * использования.
 *
 * @param newValue
 *     Новое значение для максимального количества управляемых
 *     пулом объектов, которые могут существовать одновременно.
```

```
     *     Установка в этой переменной значения,
     *     которое меньше значения, возвращаемого методом
     *     getInstanceCount, не повлечет за собой удаление объектов
     *     из пула. Просто предотвращается
     *     создание каких-либо новых объектов, управляемых пулом.
     */
    public void setMaxInstances(int newValue) {
      maxInstances = newValue;
    } // setMaxInstances()

    /**
     * Возвращает из пула объект. При пустом пуле будет создан
     * объект, если количество управляемых пулом объектов не
     * больше или равно значению, возвращаемому методом
     * getMaxInstances. Если количество управляемых пулом
     * объектов превышает это значение, то данный метод
     * возвращает null.
     */
    public Object getObject() {
      synchronized (pool) {
        Object thisObject = removeObject();
        if (thisObject!=null) {
          return thisObject;
        } // if thisObject
        if (getInstanceCount() < getMaxInstances()){
          return createObject();
        } else {
          return null;
        } // if
      } // synchronized (pool)
    } // getObject()

    /**
     * Возвращает из пула объект. При пустом пуле будет
     * создан объект, если количество управляемых пулом объектов
     * не больше или равно значению, возвращаемому методом
     * getMaxInstances. Если количество управляемых пулом объектов
     * превышает это значение, то данный метод будет ждать до тех
     * пор, пока какой-нибудь объект не станет доступным для
     * повторного использования.
     *
     * @throws InterruptedException
```

```
 *     Если вызывающий поток был прерван.
 */
public Object waitForObject() throws InterruptedException {
   synchronized (pool) {
      Object thisObject = removeObject();
      if (thisObject!=null) {
         return thisObject;
      } // if thisObject
      if (getInstanceCount() < getMaxInstances()){
         return createObject();
      } else {
           do {
           // Ожидать извещения о том, что объект был возвращен
           // в пул.
           pool.wait() ;
           thisObject = removeObject();
           } while (thisObject==null);
           return thisObject;
      } // if
   } // synchronized (pool)
} // waitForObject()

/**
 * Удаляет объект из коллекции пула и возвращает его.
 */
private Object removeObject() {
   while (pool.size()>0) {
      SoftReference thisRef
        = (SoftReference)pool.remove(pool.size()-1);
      Object thisObject = thisRef.get();
      if (thisObject!=null) {
         return thisObject;
      } // if thisObject
      instanceCount-;
   } // while
   return null;
} // removeObject()

/**
 * Создает объект, управляемый этим пулом.
```

```
     */
    private Object createObject() {
       Object newObject = creator.create();
       instanceCount++;
       return newObject;
    } // createObject()

    /**
     * Освобождает объект, помещая его в пул для повторного
     * использования.
     *
     * @param obj
     *     Объект, доступный для повторного использования.
     */
    public void release( Object obj ) {
       // no nulls
       if ( obj == null ) {
          throw new NullPointerException();
       } // if null
       if ( !poolClass.isInstance(obj)) {
          String actualClassName = obj.getClass().getName();
          throw new ArrayStoreException(actualClassName);
       } // if isInstance
       synchronized (pool) {
          pool.add(obj);
          // Известить ожидающий поток о том, что объект помещен
          // в пул.
          pool.notify();
       } // synchronized
    } // release ()
} // class SoftObjectPool
```

Теперь рассмотрим листинг класса, применяющего другую технологию, ограничивающую количество объектов в пуле. Те фрагменты кода, которые совпадают с фрагментами кода для предыдущего класса, в этом листинге опущены.

```
public class ObjectPool implements ObjectPoolIF {
   private int size ;
```

Существует определенное ограничение на количество управляемых объектов, поэтому этот класс может использовать простой массив для хранения таких

объектов. Для подсчета реального количества объектов используется переменная экземпляра с именем `size`.

```
    /**
     * Этот массив содержит объекты, которые ожидают многократного
     * использования. Им управляют, как стеком.
     */
    private Object[] pool ;
...
    /**
     * Внутренние операции синхронизируются с использованием этого
     * объекта.
     */
    private Object lockObject = new Object();
```

Более подробное объяснение объектов блокировки содержится в описании шаблона Internal Lock Object.

```
    ...
    /**
     * Constructor
     *
     * @param poolClass
     *     Класс объектов пула.
     * @param creator
     *     Объект, которому пул будет делегировать ответственность
     *     за создание управляемых им объектов.
     * @param capacity
     *     Количество неиспользуемых объектов, содержащихся
     *     в данный момент в пуле.
     * @param maxInstances
     *     Максимальное количество экземпляров класса poolClass,
     *     которым пул разрешает существовать одновременно.
     */
    public ObjectPool( Class poolClass,
                       CreationIF creator,
                       int capacity,
                       int maxInstances ) {
      size = 0;
      this.creator = creator;
      this.maxInstances = maxInstances;
      pool = (Object[])Array.newInstance(poolClass, capacity);
```

```
    } // constructor(Class, CreationIF, int, int)

    /**
     * Возвращает количество объектов в пуле.
     */
    public int getSize() {
      return size;
    } // getSize()

    /**
     * Возвращает максимальное количество объектов,
     * которые могут находиться в пуле, ожидая повторного
     * использования.
     */
  public int getCapacity() {
      return pool.length;
    } // getCapacity()

    /**
     * Задает максимальное количество объектов,
     * которые могут находиться в пуле, ожидая повторного
     * использования.
     *
     * @param newValue
     *     Новое значение для максимального количества объектов,
     *     которые могут находиться в пуле. Это значение должно
     *     быть больше нуля.
     */
    public void setCapacity(int newValue) {
      if (newValue<=0) {
        String msg = "Capacity must be greater than zero:"
                     + newValue;
        throw new IllegalArgumentException(msg);
      } // if
      synchronized (lockObject) {
        Object[] newPool = new Object[newValue];
        System.arraycopy(pool, 0, newPool, 0, newValue);
        pool = newPool;
      } // synchronized
    } // setCapacity(int)
```

```
/**
 * Возвращает из пула объект. При пустом пуле
 * будет создан объект, если количество управляемых пулом
 * объектов не больше или равно значению,
 * возвращаемому методом getMaxInstances.
 * Если количество управляемых пулом объектов превышает это
 * значение, то данный метод возвращает null.
 */
public Object getObject() {
   synchronized (lockObject) {
      if (size>0) {
         return removeObject();
      } else if (getInstanceCount() < getMaxInstances()){
         return createObject();
      } else {
         return null;
      } // if
   } // synchronized (lockObject)
} // getObject()

/**
 * Возвращает из пула объект. При пустом пуле
 * будет создан объект, если количество управляемых пулом
 * объектов не больше или равно значению, возвращаемому
 * методом getMaxInstances.
 * Если количество управляемых пулом объектов превышает это
 * значение, то данный метод будет ждать до тех пор, пока
 * какой-нибудь управляющий объект не станет доступным для
 * повторного использования.
 *
 * @throws InterruptedException
 *     Если вызывающий поток был прерван.
 */
public Object waitForObject() throws InterruptedException {
   synchronized (lockObject) {
      if (size>0) {
         return removeObject();
      } else if (getInstanceCount() < getMaxInstances()){
         return createObject();
      } else {
```

```
            do {
               // Ожидает извещения о том,
               // что объект был возвращен назад в пул.
                  wait();
            } while(size<=0);
            return removeObject();
         } // if
      } // synchronized (lockObject)
   } // waitForObject()

   /**
    * Удаляет объект из пула и возвращает его.
    */
   private Object removeObject() {
      size--;
      return pool[size];
   } // removeObject()

...

   /**
    * Освобождает объект для пула с целью его повторного
    * использования.
    *
    * @param obj
    *     Объект, доступный для повторного использования.
    * @throws ArrayStoreException
    *     Если данный объект не является экземпляром класса,
    *     переданного конструктору объекта этого пула.
    * @throws NullPointerException
    *     Если данный объект равен null.
    */
   public void release( Object obj ) {
      //no nulls
      if ( obj == null ) {
         throw new NullPointerException();
      } // if null
      synchronized (lockObject) {
         if (getSize() < getCapacity()) {
               pool[size] = obj;
```

```
                size++;
              // Оповещаем ожидающий поток о том,
              // что мы поместили объект в пул.
              lockObject.notify();
            } // if
          } // synchronized
        } // release()
      } // class ObjectPool
```

А теперь приведем листинг интерфейса `CreationIF`.

```
public interface CreationIF {
  /**
   * Возвращаем вновь созданный объект.
   */
  public Object create() ;
} // interface creationIF
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ OBJECT POOL

Cache Management. Шаблон Cache Management управляет многократным использованием определенных экземпляров класса. Шаблон Object Pool управляет и создает экземпляры класса, которые могут использоваться как взаимозаменяемые.

Factory Method. Шаблон Factory Method может использоваться для инкапсуляции логики создания объектов, однако он не управляет ими после их создания.

Singleton. Объекты, которые управляют пулами объектов, обычно являются одиночками.

Thread Pool. Шаблон Thread Pool (рассмотренный в книге [Grand99]) представляет собой специальную форму шаблона Object Pool.

Lock Object. Шаблон Lock Object может использоваться при реализации шаблона Object Pool.

ГЛАВА

Разделяющие шаблоны проектирования

Общая стратегия, призванная решать проблемы, может быть сформулирована так: *разделяй и властвуй.* Она предусматривает разбиение сложной проблемы, трудно поддающейся решению, на более простые проблемы, которые решить намного легче. Шаблоны, описанные в данной главе, можно рассматривать как руководство, помогающее разделить классы и интерфейсы таким образом, чтобы облегчить создание хорошего проекта.

Шаблон Filter описывает, как удобно организовать вычисления, выполняемые над потоком данных.

Шаблон Composite помогает в создании иерархии объектов.

Шаблон Read-Only Interface описывает способы разделения классов, использующих некоторый объект, причем те, которым разрешено изменять объект, могут это делать, а те, которым не разрешено, — не могут.

Filter (Фильтр)

Этот шаблон был ранее описан в работе [BMRSS96].

СИНОПСИС

Объекты, имеющие совместимые интерфейсы, но выполняющие различные преобразования и вычисления над потоками данных, могут динамически объединяться для выполнения произвольных операций.

КОНТЕКСТ

Существует множество программ, целью которых является выполнение вычислений над потоками данных и/или их анализ. Примером программы, совершающей простые преобразования содержимого потока данных, может служить программа `uniq` в UNIX. Она организует свои входные данные в виде строк. Обычно программа `uniq` копирует все считываемые ею строки в свои выходные данные. Однако если она обнаруживает последовательные строки, содержащие идентичные символы, то она копирует в свои выходные данные только первую такую строку. В UNIX есть также программа `wc`, которая выполняет простой анализ потока данных. Она вычисляет количество символов, слов и строк в потоке данных. Компиляторы выполняют сложные серии преобразований и анализируют на входе исходный код с целью получения на выходе двоичных данных.

Поскольку многие программы выполняют преобразования и анализ потоков данных, должна быть несомненная польза в том, чтобы определить классы, выполняющие наиболее распространенные преобразования и анализ. Такие классы будут использоваться очень часто.

Классы, выполняющие простые преобразования и анализ потоков данных, по своей сути должны быть весьма универсальными. При написании таких классов невозможно предвидеть все случаи их использования. Некоторые приложения захотят применять преобразования и анализ только для определенной части потока данных. Очевидно, что эти классы должны быть написаны так, чтобы допускать большую гибкость объединения их экземпляров друг с другом. Один из способов осуществления такой гибкости предполагает задание общего интерфейса для всех этих классов таким образом, чтобы экземпляр одного мог использовать экземпляр другого, не заботясь о том, экземпляром какого класса является объект (рис. 6.1).

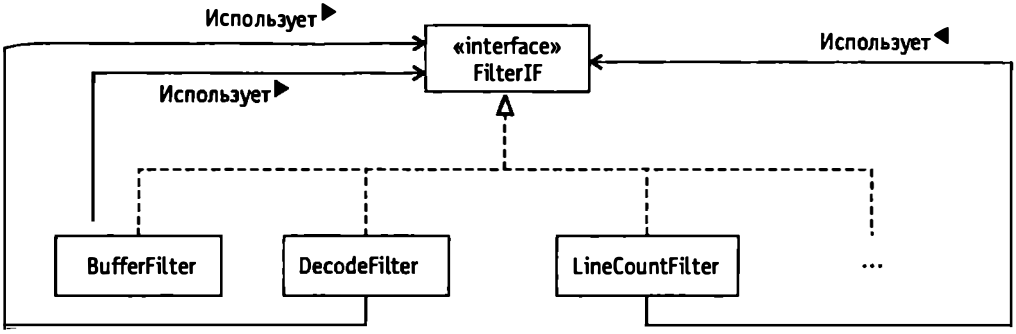


Рис. 6.1. Фильтры данных

МОТИВЫ

☺ Классы, которые реализуют общие преобразования и анализ данных, могут использоваться самыми разнообразными программами.

☺ Должна существовать возможность динамического комбинирования объектов анализа данных и преобразования посредством их объединения.

☺ Использование объектов преобразования/анализа должно быть прозрачным для других объектов.

РЕШЕНИЕ

Решение основано на использовании общих интерфейсов и делегирования. Шаблон Filter организует свои классы в виде источников, приемников и фильтров данных. Классы фильтров данных выполняют операции преобразования и анализа. Существуют две основные формы шаблона Filter:

- потоки данных, получаемые объектом приемника данных при вызове методов источника данных. Эта форма шаблона Filter называется pull-формой;
- потоки данных, образующиеся в том случае, когда объект источника передает данные методу объекта приемника. Эту форму шаблона Filter называют push-формой.

На рис. 6.2 представлена pull-форма шаблона Filter.

Опишем классы, участвующие в pull-форме шаблона Filter.

SourceIF. Интерфейс, выступающий в этой роли, объявляет один или несколько методов, которые в ответ на их вызов возвращают данные. На рис. 6.2 один такой метод имеет имя `getData`.

Source. Класс, выступающий в этой роли, отвечает главным образом за предоставление данных, а не за их преобразование или анализ. Классы, играющие эту роль, должны реализовывать интерфейс `SourceIF`.

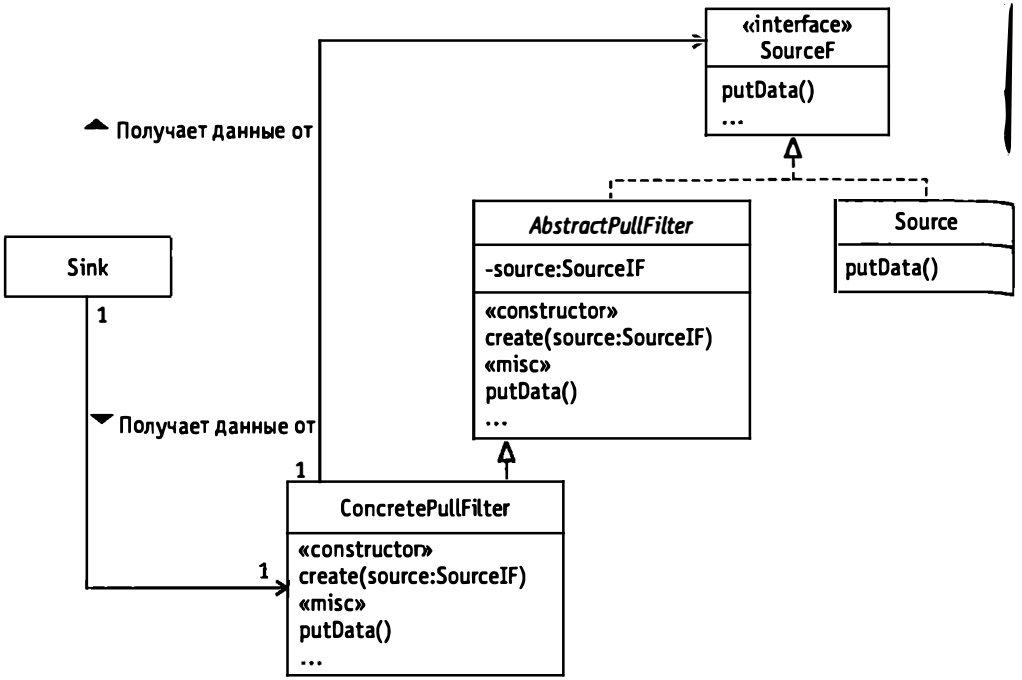


Рис. 6.2. Pull-форма шаблона Filter

AbstractPullFilter. Класс, выступающий в этой роли, представляет собой абстрактный суперкласс для классов, которые преобразуют и анализируют данные. Он имеет конструктор, в который передается объект `SourceIF`. Экземпляры этого класса делегируют выборку данных объекту `SourceIF`, который был передан их конструктору.

Классы `AbstractPullFilter` обычно имеют переменную, которая ссылается на объект `SourceIF`, переданный их конструктору. Однако, чтобы обеспечить независимость их подклассов от этой переменной экземпляра, она должна быть закрытой. Классы `AbstractPullFilter` обычно определяют метод `getData`, который просто вызывает метод `getData` объекта `SourceIF`, на который ссылается переменная экземпляра.

ConcretePullFilter. Класс, выступающий в этой роли, представляет собой конкретный подкласс класса `AbstractPullFilter`. Они замещают унаследованный ими метод `getData` с целью выполнения соответствующих операций преобразования или анализа.

Sink. Экземпляры классов, выступающих в этой роли, вызывают метод `getData` объекта `SourceIF`. В отличие от объектов `ConcretePullFilter` экземпляры классов `Sink` используют данные, не передавая их другому объекту `AbstractPullFilter`.

На рис. 6.3 представлена диаграмма классов для push-формы шаблона Filter.

Рассмотрим роли, выполняемые этими классами и интерфейсом.

SinkIF. Интерфейс, выступающий в этой роли, объявляет один или несколько методов, которые получают данные с помощью одного из параметров. На рис. 6.3 один такой метод имеет имя `putData`.

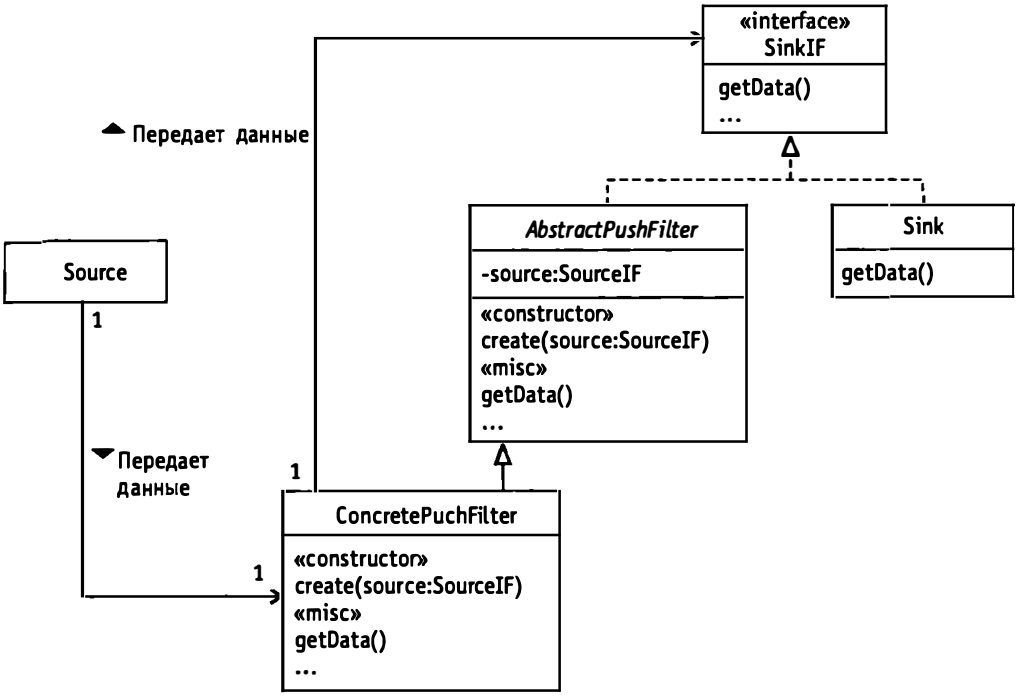


Рис. 6.3. Push-форма шаблона Filter

Sink. Класс, выступающий в этой роли, отвечает главным образом за получение и обработку данных, а не за их преобразование или анализ. Классы, играющие эту роль, должны реализовывать интерфейс `SinkIF`. Данные передаются объектам `Sink` посредством их передачи методу `putData` этого объекта.

AbstractPushFlter. Класс, выступающий в этой роли, представляет собой абстрактный суперкласс для классов, которые преобразуют и/или анализируют данные. Он имеет конструктор, которому передается объект `SinkIF`. Экземпляры этого класса передают данные объекту `SinkIF`, который был передан их конструктору. Поскольку подклассы этого класса реализуют интерфейс `SinkIF`, их экземпляры могут принимать данные от других объектов, которые передают данные объектам `SinkIF`.

Классы `AbstractPushFilter` обычно имеют переменную экземпляра, которая ссылается на объект `SinkIF`, переданный их конструктору. Однако, чтобы обеспечить независимость их подклассов от этой переменной экземпляра, она должна быть закрытой. Классы `AbstractPushFilter` обычно определяют

метод putData, вызывающий метод putData объекта SinkIF, на который ссылается переменная экземпляра.

ConcretePushFilter. Класс, выступающий в этой роли, представляет собой конкретный подкласс класса AbstractPushFilter. Они замещают унаследованный ими метод putData с целью выполнения соответствующих операций преобразования или анализа.

Source. Экземпляры классов, выступающих в этой роли, вызывают метод putData объекта SinkIF.

РЕАЛИЗАЦИЯ

Классы фильтров должны быть реализованы так, чтобы они ничего не знали о программах или о других классах фильтров, вместе с которыми они будут использоваться. Из этого следует, что объекты фильтров должны общаться друг с другом только через данные, которыми они обмениваются.

Создание классов фильтров независимыми от программы, в которой они используются, повышает вероятность их многократного использования. Однако в некоторых случаях эффективность снижается, если фильтр не использует информацию, зависящую от контекста. Лучший выход в такой ситуации — компромисс. Например, можно определить один или несколько интерфейсов, которые объявляют методы для предоставления объекту фильтра информации, зависящей от контекста. Если программа обнаруживает, что объект фильтра реализует один из таких интерфейсов, она может использовать этот интерфейс для предоставления фильтру дополнительной информации.

Заставляя методы putData или getData класса фильтра проходить через методы getData или putData их суперкласса, мы добавляем некоторые дополнительные издержки. Эти дополнительные сложности совсем несложно устранить. Если ваши классы работают на JVM, которая применяет технологию HotSpot компании Sun, то предлагаемая HotSpot оптимизация устранит почти все дополнительные издержки. Если ваши классы работают в среде, в которой не предусматривается оптимизация такого рода, существуют компиляторы, которые могут выполнить оптимизацию посредством встраивания (in-lining) обращений к методам getData или putData суперкласса.

СЛЕДСТВИЯ

☺ Часть программы, которая соответствует шаблону Filter, может быть разбита на набор источников, приемников и фильтров.

☺ Объекты-фильтры, не сохраняющие внутреннее состояние, могут быть динамически заменены на стадии выполнения программы. Это свойство фильтров, не хранящих состояния, позволяет динамически изменять поведение и адаптировать его в соответствии с различными требованиями на стадии выполнения программы.

- Программа вполне может использовать обе формы шаблона Filter. Однако один и тот же класс обычно не участвует в обеих формах этого шаблона.
- Если проект предусматривает динамическое добавление или удаление фильтров во время обработки потока данных, то придется создать механизм, позволяющий полностью управлять подобными изменениями.

ПРИМЕНЕНИЕ В JAVA API

Пакет `java.io` содержит класс `FilterReader`, который принимает участие в шаблоне Filter в качестве абстрактного класса-фильтра источника. Соответствующий абстрактный класс источника называется `Reader`. Конкретные подклассы класса `FilterReader` — это `BufferedReader`, `FileReader` и `LineNumberReader`. Не существует отдельного интерфейса, который бы выполнял роль `SourceIF`. Эту роль взял на себя тот же класс `Reader`.

Пакет `java.io` содержит класс `FilterWriter`, который участвует в шаблоне Filter как абстрактный класс-фильтр приемника. Соответствующий абстрактный класс приемника — это `Writer`. Конкретные подклассы класса `FilterWriter` — `BufferedWriter`, `FileWriter` и `PrintWriter`. Кроме того, класс `Writer` исполняет также роль `SinkIF`.

Приведем обычную организацию объектов класса `FilterReader` в программе, которая читает строки текста (команды) и отслеживает номера строк для выдачи сообщений об ошибках:

```
LineNumberReader in;
void init(String fName) {
   FileReader fin;
      try {
         fin = new FileReader(fName);
         in = new LineNumberReader(new BufferedReader(fin));
      } catch (FileNotFoundException e) {
         System.out.println("Unable to open "+fName);
         ...
      }
   ...
```

ПРИМЕР КОДА

В качестве примера классов, реализующих pull-форму шаблона Filter, опишем классы, которые читают и фильтруют байтовые потоки. Первым приведем класс, участвующий в шаблоне Filter в качестве абстрактного источника:

```
public interface InStreamIF {
   /**
    * Прочитать байты из байтового потока и записать их в массив.
```

```
     * @param array Заполняемый массив.
     * @return Если байтов недостаточно для заполнения
     *     массива, то этот метод  завершается, заполнив массив
     *     реальным количеством байтов. Возвращает общее количество
     *     байтов или -1, если достигнут конец потока данных.
     * @throws IOException Если появляется ошибка ввода-вывода.
    public int read(byte[] array) throws IOException;
} // interface InStreamIF
```

А теперь приведем класс, который реализует интерфейс `InStreamIF` и участвует в шаблоне Filter в качестве источника:

```
/**
* Этот класс считывает поток байтов из файла.
*/
public class FileInStream implements InStreamIF {
   private RandomAccessFile file;

   /**
    * Constructor
    * @param fName Имя считываемого файла.
    */
   public FileInStream(String fName) throws IOException {
      file = new RandomAccessFile(fName, "r");
   } // Constructor(String)

   /**
    * Считывает байты из файла и заполняет этими байтами массив.
    */
   public int read(byte[] array) throws IOException {
      return file.read(array);
   } // read (byte [])
} // class FileInStream
```

Следующий класс принимает участие в шаблоне Filter в качестве фильтра абстрактного источника:

```
public abstract class FilterInStream implements InStreamIF {
   private InStreamIF inStream;
   /**
    * Constructor
    * @param inStream
    *     Объект InStreamIF, которому данный объект должен
    *     делегировать операции чтения.
    */
```

```
    public FilterInStream(InStreamIF inStream)
              throws IOException {
       this.inStream = inStream;
    } // Constructor(InStreamIF)

    /**
     * Считывает байты из байтового потока и заполняет ими массив.
     */
    public int read(byte[] array) throws IOException {
       return inStream.read(array);
    } // read(byte[])
} // class FilterInStream
```

А теперь рассмотрим классы, которые участвуют в шаблоне Filter в качестве фильтра конкретного источника. Одни из них выполняют простой анализ, который состоит в том, что они подсчитывают количество считанных байтов:

```
public class ByteCountInStream extends FilterInStream {
    private long byteCount = 0;

    /**
     * Constructor
     * @param inStream
     *     InStream, которому данный объект должен делегировать
     *     операции чтения.
     */
    public ByteCountInStream(InStreamIF inStream)
              throws IOException {
       super(inStream);
    } // Constructor(InStream)

    /**
     * Считывает байты из байтового потока в массив.
     */
    public int read(byte[] array) throws IOException {
       int count;
       count = super.read(array);
       if (count >0)
          byteCount += count;
       return count;
    } // read(byte[])
```

```
    /**
     * Возвращает количество байтов, считанных этим объектом.
     */
    public long getByteCount() {
       return byteCount;
    } // getByteCount()
} // class ByteCountInStream
```

И наконец, приведем класс-фильтр, который выполняет преобразование кодов символов потока байтов:

```
/**
* Этот класс рассматривает каждый байт байтового потока
* как восьмибитовый код символа и преобразует его в другой
* код символа, используя для этой цели таблицу
* преобразований.
*/
public class TranslateInStream extends FilterInStream {
    private byte[] translationTable;
    private final static int TRANS_TBL_LENGTH = 256;

    /**
    * Constructor
    * @param inStream
    *     Объект InStreamIF, которому данный объект должен
    *     делегировать операции чтения.
    * @param table
    *     Массив байтов, используемый с целью определения значений
    *     преобразования для символьных кодов. Меняет символ
    *     с кодом n на n-й элемент таблицы преобразования.
    *     Если длина массива превышает количество элементов,
    *     задаваемое TRANS_TBL_LENGTH, то дополнительные элементы
    *     игнорируются. Если массив меньше TRANS_TBL_LENGTH
    *     элементов, то преобразование не производится для тех
    *     символов, коды которых больше или равны длине массива
    *     table.
    */
public TranslateInStream(InStreamIF inStream,
                byte[] table) throws IOException{
  super(inStream);
       // Создает таблицу преобразования путем копирования
       // данных преобразования.
```

```
    translationTable = new byte[TRANS_TBL_LENGTH];
    System.arraycopy(table, 0, translationTable, 0,
              Math.min(TBANS_TBL_LENGTH,
                 table.length));
    for (int i = table.length; i < TRANS_TBL_LENGTH; i++){
       translationTable[i] = (byte)i;
    } // for
    } // Constructor(InStream)

  public int read(byte[] array) throws IOException {
    int count;
    count = super.read(array);
    for (int i = 0; i < count; i++) {
       array[i] = translationTable[array[i]];
    } // for
    return count;
    } // read(byte[])
  } // class ByteCountInStream
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ FILTER

Composite. Шаблон Composite может служить альтернативой шаблону Filter. Он позволяет представлять данные в виде древовидных структур.

Pipe. Шаблон Pipe иногда выступает в качестве альтернативы шаблону Filter, а иногда используется вместе с ним.

Описание шаблона Pipe приводится в работе [BMRSS96]. Подобно шаблону Filter, шаблон Pipe позволяет объекту, представляющему источник данных, отправлять поток данных объекту, представляющему приемник данных. Вместо перемещения данных, инициируемых объектом источника или приемника, они действуют асинхронно по отношению друг к другу. Объект источника записывает данные в буфер, когда захочет. Приемник считывает данные из буфера, когда захочет. Если буфер пустой и приемник пытается считать из него данные, то приемник ожидает появления данных в буфере.

Java API содержит классы `java.io.PipedReader` и `java.io.PipedWriter`, которые вместе реализуют шаблон Pipe.

Decorator. Шаблон Filter — специальный случай шаблона Decorator, в котором объект источника или приемника данных достраивается с целью добавления логики для управления потоком данных.

Composite (Компоновщик)

Шаблон Composite известен также как шаблон Recursive Composition (Рекурсивная композиция). Ранее он был описан в работе [GoF95].

СИНОПСИС

Шаблон Composite позволяет создавать сложные объекты посредством рекурсивного объединения похожих объектов в виде дерева. Кроме того, этот шаблон дает возможность согласованно управлять объектами дерева, требуя от всех объектов наличия общего интерфейса или суперкласса.

Шаблон Composite представлен в книге с точки зрения рекурсивного построения сложного объекта, состоящего из других объектов. Composite относится к разделяющим шаблонам, поскольку в процессе создания проекта этот шаблон часто используется для рекурсивного разложения составного объекта на более простые объекты.

КОНТЕКСТ

Предположим, что создается программа форматирования документа. Она форматирует символы, представляя их в виде строк текста, которые образуют колонки, разделенные на страницы. Однако документ может содержать и другие элементы. Колонки и страницы могут иметь рамки с колонками внутри. Колонки, рамки и строки текста могут содержать рисунки (рис. 6.4).

Очевидно, что сложностей здесь достаточно. Объекты `Page` и `Frame` должны знать, как обрабатывать и комбинировать элементы двух видов. Объекты `Column` должны знать, как обрабатывать и комбинировать элементы трех видов. Шаблон Composite устраняет данную сложность, позволяя таким объектам управлять элементами только одного вида. Этого можно достичь, если классы всех элементов документа будут реализовывать общий интерфейс. На рис. 6.5 показано, как можно упростить отношения между классами элементов документа при помощи шаблона Composite.

При использовании шаблона Composite вводится общий интерфейс для всех элементов документа и общий суперкласс для всех контейнерных классов. При этом количество отношений агрегации уменьшается до одного. Теперь управление агрегацией — обязанность класса `CompositeDocumentElement`. Конкретные контейнерные классы (`Document`, `Page`, `Column` и др.) должны знать только, как соединять элементы одного вида.

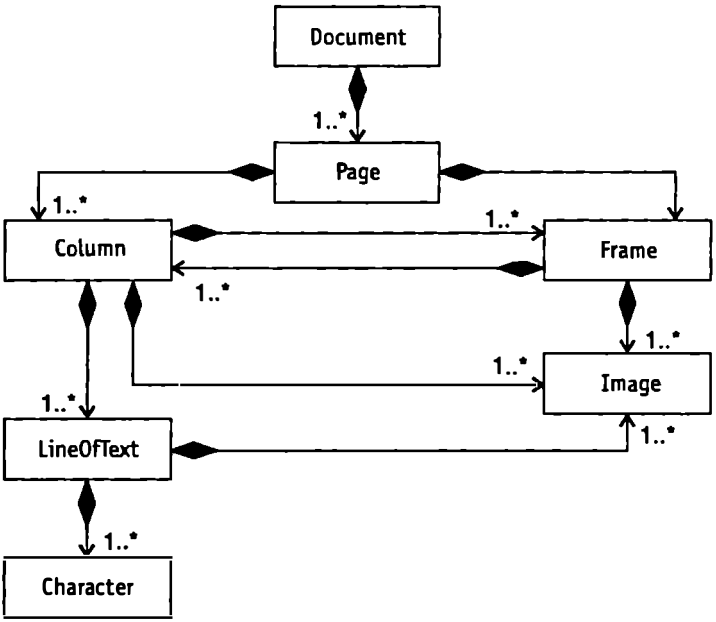


Рис. 6.4. Отношения между контейнерными классами документа

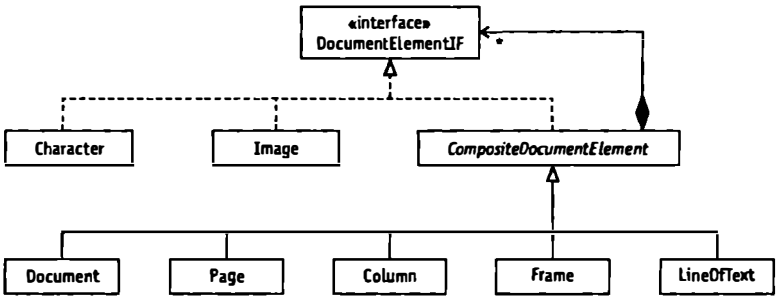


Рис. 6.5. Документ, использующий шаблон Composite

ОТИВЫ

Есть сложный объект, который нужно представить в виде иерархии объектов, представляющей отношение «часть — целое».

Нужно свести к минимуму сложность иерархии «часть — целое», оставляя минимальным количество различных дочерних объектов, о которых должны быть осведомлены объекты дерева.

Не предъявляется никаких требований к большей части объектов из иерархии по их различию.

РЕШЕНИЕ

Свести к минимуму сложность составного объекта, организованного в виде иерархии «часть — целое». Это выполняется путем предоставления интерфейса, который должен реализовываться всеми объектами, входящими в иерархию, и абстрактного суперкласса для всех составных объектов этой иерархии (рис. 6.6).

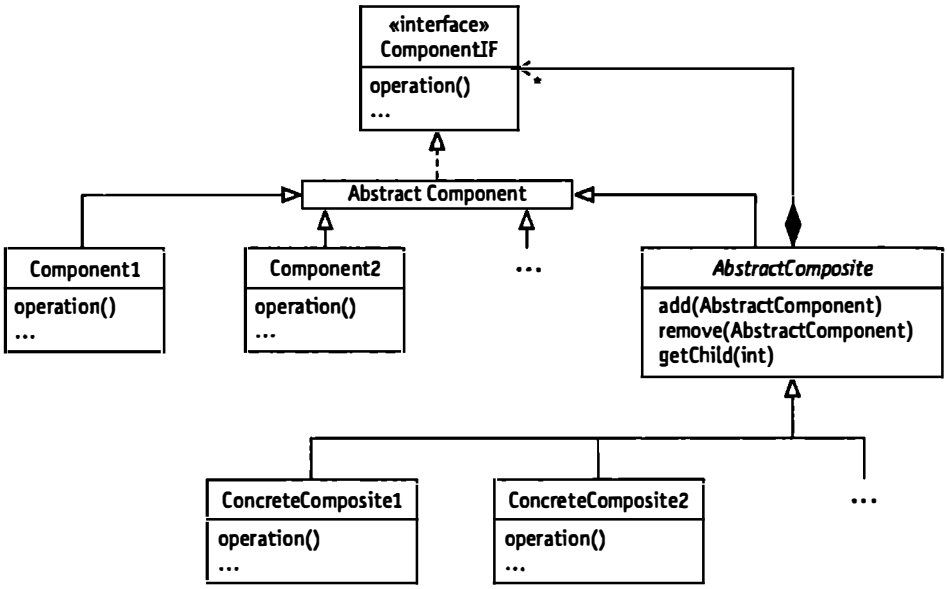


Рис. 6.6. Отношения между классами составного объекта

Опишем интерфейс и классы, принимающие участие в шаблоне Composite.

ComponentIF. Интерфейс, выступающий в этой роли, реализуется всеми объектами, входящими в иерархию и образующими составной объект. Как правило, составные объекты рассматривают содержащиеся в них объекты как экземпляры классов, реализующих интерфейс `ComponentIF`, а не как экземпляры некоторых реально существующих классов.

Component1, Component2 и т.д. Экземпляры этих классов считаются листьями дерева в древовидной структуре.

AbstractComposite. Класс, выступающий в этой роли, представляет собой абстрактный суперкласс для всех составных объектов, принимающих участие в шаблоне Composite. `AbstractComposite` определяет и предоставляет реализации методов, задаваемых по умолчанию и предназначенных для управления компонентами составного объекта. Метод `add` добавляет компонент в составной объект, а метод `remove` — удаляет. Метод `getChild` возвращает ссылку на объект компонента составного объекта.

ConcreteComposite1, ConcreteComposite2 и т.д. Экземпляры этих классов представляют собой составные объекты, которые используют другие экземпляры класса `AbstractComponent`.

Экземпляры этих классов могут быть объединены и представлены в виде связного дерева (рис. 6.7).

Обратите внимание, что не нужно иметь абстрактный класс для составного объекта, если существует только один конкретный класс составного объекта.

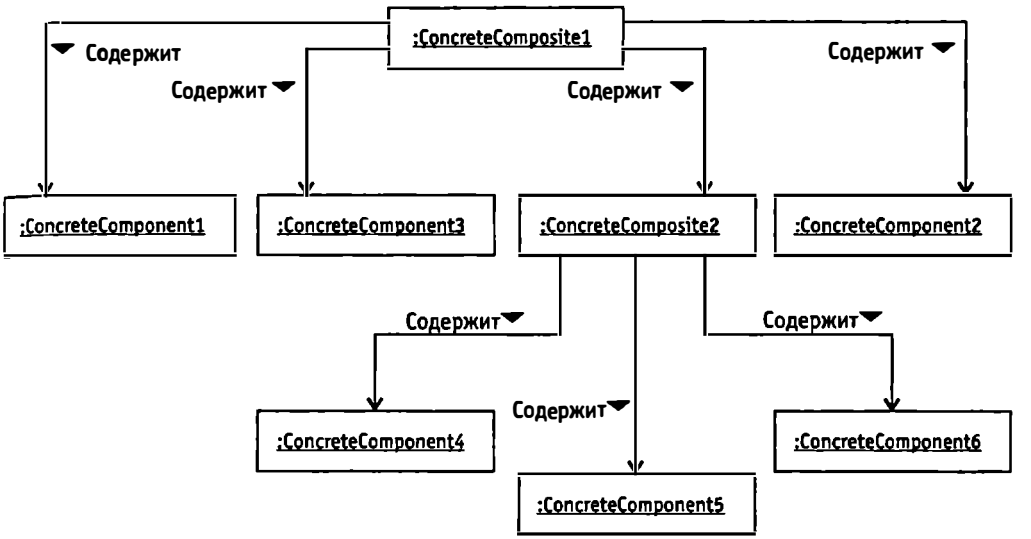


Рис. 6.7. Составной объект класса ConcreteComposite1

РЕАЛИЗАЦИЯ

Если классы, участвующие в шаблоне Composite, реализуют какие-либо операции, делегируя их своим родительским объектам, то для повышения производительности и простоты использования можно добавить в каждый экземпляр класса `AbstractComponent` ссылку на своего родителя. При реализации операции получения ссылки на предка важно сделать это так, чтобы обеспечить согласованность между предком и потомком. Объект `ComponentIF` идентифицирует объект `AbstractComposite` как своего родителя тогда и только тогда, когда `AbstractComposite` идентифицирует его как одного из своих потомков. Наилучший способ реализации этого механизма состоит в изменении ссылок на родителей и потомков только при помощи методов добавления и удаления, задаваемых в классе `AbstractComposite`.

Совместное использование компонентов многими родителями при помощи шаблона Flyweight (см. гл. 7) — это способ сохранения памяти. Однако совме-

стно используемым компонентам сложно поддерживать ссылки на родителей надлежащим образом.

Класс `AbstractComposite` может предоставлять составным объектам задаваемую по умолчанию реализацию управления потомком. Следует знать, однако, что очень часто классы конкретных составных объектов замещают задаваемую по умолчанию реализацию.

Если конкретный составной объект делегирует операцию входящим в его состав объектам, то кэширование результата операции может улучшать производительность. Если конкретный составной объект кэширует результат операции, очень важно, чтобы объекты, образующие составной объект, уведомляли последний о некорректности некоторых данных, хранящихся в кэше.

СЛЕДСТВИЯ

☺ Можно получить доступ к составному объекту, структурированному в виде дерева, и образующим это дерево объектам через интерфейс `ComponentIF` независимо от того, простыми или составными объектами они являются. Структура составного объекта не вынуждает другие объекты делать подобное разграничение.

☺ Клиентские объекты класса `AbstractComponent` могут рассматривать его просто как класс `AbstractComponent`, не заботясь о том, с каким подклассом они имеют дело.

☺ Если клиент вызывает метод объекта `ComponentIF`, который, как ожидается, должен выполнить некоторую операцию, и объект `ComponentIF` является объектом `AbstractComposite`, то `AbstractComposite` может делегировать выполнение этой операции объектам `ComponentIF`, которые входят в него. Точно так же если клиентский объект вызывает метод объекта `ComponentIF`, который не является `AbstractComposite`, и метод нуждается в некоторой контекстной информации, то объект `ComponentIF` делегирует запрос на получение контекстной информации своему родителю.

- Некоторые компоненты могут реализовывать специфические для этого компонента операции. Например, в разделе «Контекст» описания данного шаблона представлен проект рекурсивной композиции документа. На его самом нижнем уровне находится документ, который состоит из элементов символов и изображений. Логично иметь метод `getFont` для символьных элементов документа. Элементы изображений документа не нуждаются в методе `getFont`. Основное удобство шаблона Composite заключается в том, что он позволяет клиентам составного объекта и содержащимся в нем объектам ничего не знать об определенном классе объектов, с которым они имеют дело. Чтобы позволить другим классам вызывать `getFont`, не имея информации об определенном классе, с которым они имеют дело, все объекты, входящие в состав документа, могут наследовать метод `getFont` от `DocumentElementIF`. В целом, при использовании шаблона `Composite`

интерфейс, выступающий в роли ComponentIF, объявляет специальные методы в том случае, если они нужны какому-либо классу ConcreteComponent.

Принцип объектно-ориентированного проектирования заключается в том, что специальные методы должны находиться только в тех классах, которым они нужны. Как правило, класс должен иметь методы, обеспечивающие связанную функциональность и образующие сцепленный набор. В этом принципе заключена суть шаблона Low Coupling/High Cohesion, описанного в книге [Grand99]. Помещение специального метода в класс общего назначения, а не в специальный класс, которому этот метод нужен, противоречит принципу сильного сцепления, поскольку при этом добавляется метод, не связанный с другими методами общего класса. Несвязанный метод наследуется подклассами общего класса, которые вообще никак не связаны с этим методом.

Поскольку простота (ввиду игнорирования сведений о классе) лежит в основе шаблона Composite, то при его использовании нет ничего плохого в том, чтобы пожертвовать сильным сцеплением ради простоты. Такое исключение из широко распространенного правила основано скорее на практике, чем на теории.

☹ Шаблон Composite позволяет любому объекту ComponentIF быть потомком AbstractComposite. Если нужно задать более строгое отношение, необходимо добавить в класс AbstractComposite или его подклассы код, который будет осведомлен о типе объекта. При этом несколько снижается ценность шаблона Composite.

ПРИМЕНЕНИЕ В JAVA API

В пакете java.awt содержится удачный пример шаблона Composite. Его класс Component играет роль ComponentIF, а класс Container выполняет роль AbstractComposite. У него есть классы, выступающие в роли ConcreteComponent, включая Label, TextField и Button. Роль ConcreteComposite играют классы Panel, Frame и Dialog.

ПРИМЕР КОДА

Применение шаблона Composite рассмотрим на примере задачи, описанной в разделе «Контекст». На рис. 6.8 представлена подробная диаграмма классов.

На рис. 6.8 изображены некоторые методы, не указанные на рис. 6.6. Из следующего кода станет понятно, что метод setFont может служить примером взаимодействия с объектом родителя. Метод getCharLength собирает информацию у потомков объекта и сохраняет ее в кэше для дальнейшего использования. Метод changeNotification используется для синхронизации сохраненной в кэше информации.

Абстрактный класс AbstractDocumentElement содержит общую логику управления шрифтами и родительским объектом.

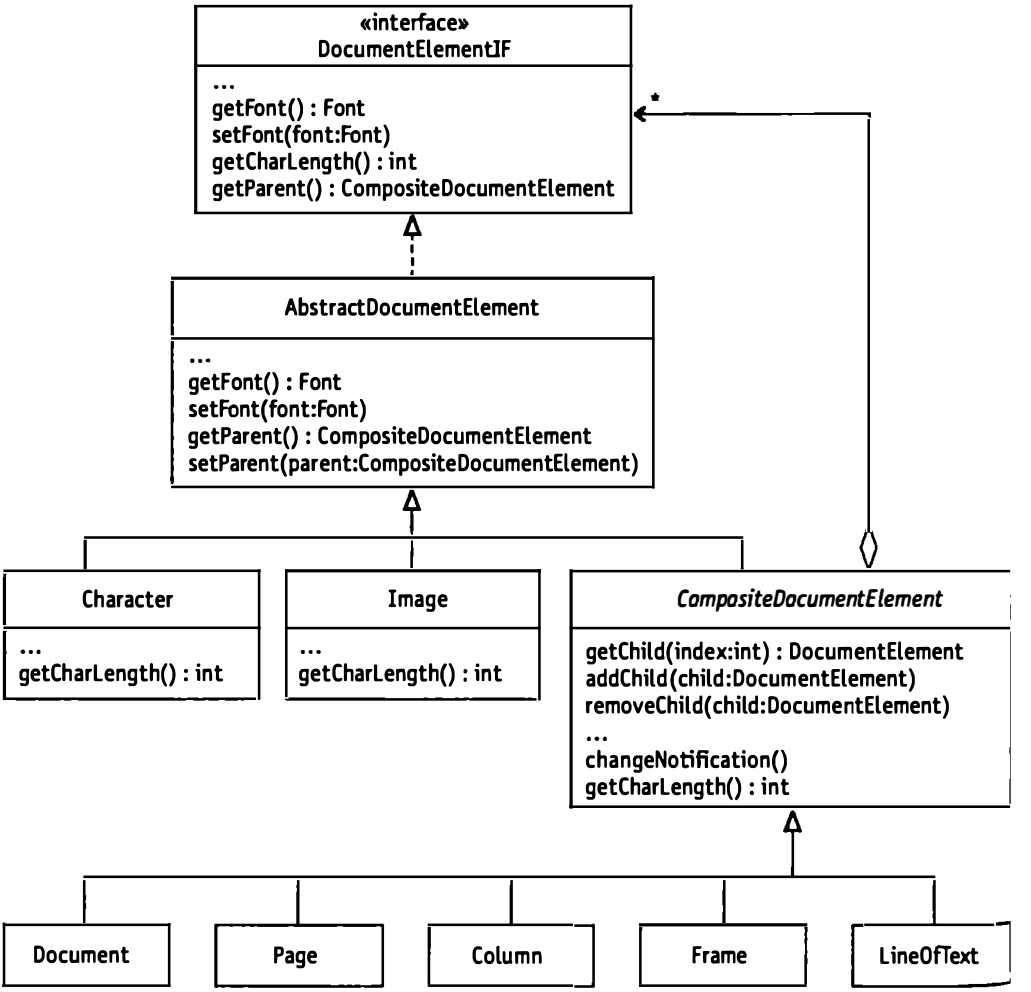


Рис. 6.8. Детализированная диаграмма классов

Рассмотрим код для интерфейса `DocumentElementIF`, который должен реализовываться всеми классами, образующими документ:

```
public interface DocumentElementIF {
   ...

   /**
    * Возвращает родителя этого объекта или null, если родитель
    * отсутствует.
    */
   public CompositeDocumentElement getParent() ;
```

```
    /**
     * Возвращает шрифт, связанный с данным объектом.
     */
    public Font getFont() ;

    /**
     * Связывает шрифт с этим объектом.
     * @param font Шрифт, связываемый с этим объектом.
     */
    public void setFont(Font font) ;

    /**
     * Возвращает количество символов, содержащихся в этом объекте.
     */
    public int getCharLength() ;
} // interface DocumentElementIF
```

Теперь листинг класса `AbstractDocumentElement`, который содержит общую логику по управлению шрифтами и родительским объектом:

```
abstract class AbstractDocumentElement
            implements DocumentElementIF {
    /**
     * Это шрифт, связанный с данным объектом.
     * Если эта переменная — null,
     * то шрифт этого объекта наследуется от предка.
     */
    private Font font;

    /**
     * Контейнер этого объекта.
     */
    private CompositeDocumentElement parent;

    ...

    /**
     * Возвращает родителя этого объекта или null, если у него нет
     * родителя.
     */
    public CompositeDocumentElement getParent() {
```

```
        return parent;
    } // getParent()

    /**
     * Задает родителя этого объекта.
     */
    protected void setParent(CompositeDocumentElement parent) {
        this.parent = parent;
    } // setParent(AbstractDocumentElement)

    /**
     * Возвращает Font, связанный с данным объектом.
     * Если нет шрифта, связанного с данным объектом,
     * то возвращает шрифт, связанный с родителем этого объекта.
     * Если нет шрифта, связанного с родителем данного объекта, то
     * возвращает null.
     */
    public Font getFont() {
        if (font != null)
            return font;
        else if (parent != null)
            return parent.getFont();
        else
            return null;
    } // getFont()

    /**
     * Связывает Font с этим объектом.
     * @param font
     *     Шрифт, который связывается с этим объектом.
     */
    public void setFont(Font font) {
        this.font = font;
    } // setFont(Font).

    /**
     * Возвращает количество символов, содержащихся в этом
     * объекте.
     */
    public abstract int getCharLength();
} // class AbstractDocumentElement
```

Код для класса CompositeDocumentElement, который представляет собой абстрактный суперкласс для всех элементов документа, содержащих другие элементы документа:

```
public abstract class CompositeDocumentElement
    extends AbstractDocumentElement {
  // Коллекция потомков этого объекта.
  private Vector children = new Vector();

  /**
   * Значение, помещенное в кэш, после предыдущего вызова
   * метода getCharLength, или -1, если charLength
   * не содержит помещенного в кэш значения.
   */
  private int cachedCharLength = -1;

  /**
   * Возвращает объект потомка этого объекта,
   * находящегося в данной позиции.
   * @param index
   *     Индекс потомка.
   */
  public DocumentElementIF getChild(int index) {
    return (DocumentElementIF)children.elementAt(index);
  } // getChild(int)

  /**
   * Делает данный DocumentElementIF потомком этого объекта.
   */
  public
  synchronized void addChild(DocumentElementIF child) {
    synchronized (child) {
      children.addElement(child);
      ((AbstractDocumentElement)child).setParent(this);
      changeNotification();
    } // synchronized
  } // addChild(DocumentElementIF)
```

Оба метода addChild и removeChild являются синхронизированными и, кроме того, содержат синхронизирующий блок для блокировки данного потомка. Это объясняется тем, что указанные методы изменяют и контейнер, и его потомка.

```
/**
 * Сделаем так, чтобы DocumentElementIF НЕ был потомком
 * данного объекта.
 */
public synchronized
void removeChild(AbstractDocumentElement child) {
   synchronized (child) {
      if (this == child.getParent()) {
          child.setParent(null) ;
      } // if
      children.removeElement(child);
      changeNotification();
   } // synchronized
} // removeChild(AbstractDocumentElement)

...

/**
 * Вызов этого метода означает, что один из потомков
 * был изменен таким образом, что он
 * делает недействительными все данные о потомках, которые
 * этот объект может помещать в кэш.
 */
public void changeNotification() {
   cachedCharLength = -1;
   if (getParent() != null)
      getParent().changeNotification();
} // changeNotification()

/**
 * Возвращает количество символов, содержащихся в этом
 * объекте.
 */
public int getCharLength() {
   int len = 0;
   for (int i = 0; i < children.size(); i++) {
      AbstractDocumentElement thisChild;
      thisChild
         = (AbstractDocumentElement)children.elementAt(i);
      len += thisChild.getCharLength();
```

```
        } // for
        cachedCharLength = len;
        return len;
    } // getCharLength()
} // class CompositeDocumentElement
```

Класс Image может служить примером класса, реализующего некоторый метод, который позволяет другим классам, образующим документ, не знать о классе Image то, что он может нуждаться в некоторой специальной обработке. Его метод getCharLength всегда возвращает 1, поэтому изображение может рассматриваться просто как большой символ.

```
class Image extends Abstract DocumentElement {
...
   public int getCharLength() {
      return 1;
   } // getCharLength()
} // class Image
```

Другие классы, представленные на диаграмме классов и являюшиеся подклассами класса CompositeDocumentElement, не содержат каких-либо свойств, которые могут быть интересны при рассмотрении шаблона Composite. Для краткости просто приведем один класс, который выглядит примерно так:

```
class Page extends CompositeDocumentElement {
...
} // class Page
```

Шаблоны проектирования, связанные с шаблоном Composite

Chain of Responsibility. Шаблон Chain of Responsibility может использоваться вместе с шаблоном Composite посредством добавления звеньев «потомок — родитель» таким образом, что потомки могут получать информацию от предка, не зная, от какого предка она получена.

Low Coupling/High Cohesion. Шаблон Low Coupling/High Cohesion (описанный в книге [Grand99]) не рекомендует помещать специальные методы в классы общего назначения, что иногда делает шаблон Composite.

Visitor. Можно использовать шаблон Visitor для инкапсуляции в одном классе операций, которые в противном случае были бы распределены по множеству

Read-Only Interface (Интерфейс, предназначенный только для чтения)

Впервые шаблон Read-Only Interface был описан в работе [LL01].

СИНОПСИС

Необходимо, чтобы объект изменялся только некоторыми его клиентами. Шаблон Read-Only Interface гарантирует, что клиенты, которым не позволено изменять объект, не будут его изменять благодаря тому, что они получают доступ к этому объекту через интерфейс, который не содержит каких-либо методов, способных изменить объект.

КОНТЕКСТ

Создается ПО, которое станет частью системы безопасности некоторого здания. В задачу входит отслеживание состояния физических сенсоров, которые фиксируют открытие и закрытие дверей, температуру в комнатах и другие параметры. Сенсоры отправляют сообщения компьютеру, на котором будет запущена разрабатываемая программа обеспечения безопасности. При получении сообщения программа обеспечения безопасности может обновлять экран или производить другие действия. Подобные взаимодействия показаны на рис. 6.9.

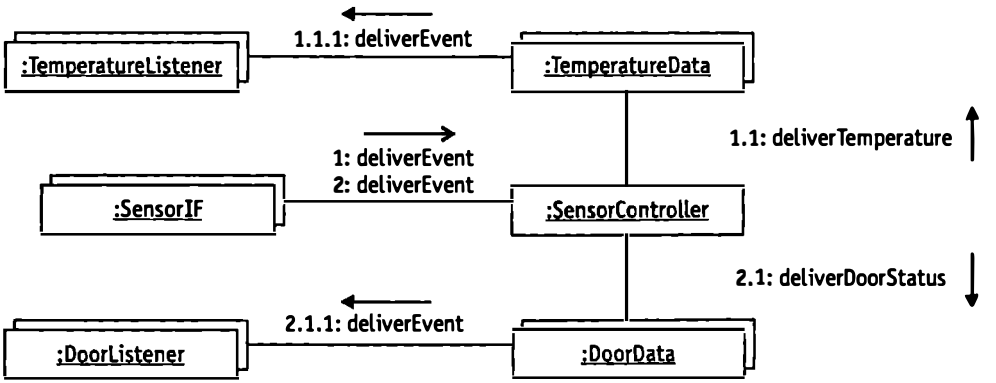


Рис. 6.9. Взаимодействие контроллера безопасности

Оба взаимодействия, представленные на данном рисунке, инициируют сенсор, программной реализацией которого является интерфейс `SensorIF`. В ходе первого взаимодействия температурный сенсор передает значение температуры

объекту `SensorController` контролирующей программы. Во втором взаимодействии сенсор передает изменение состояния двери (открыта, закрыта, заблокирована и т.д.). Все сенсоры передают данные объекту `SensorController`. Объект `SensorController` отвечает за идентификацию и определение типа сенсора, отправившего данные. Он передает данные тому объекту контролирующей программы, который соответствует сенсору, отправившему данные.

Каждый объект данных, например, объекты `DoorData` и `TemperatureData`, представленные на рис. 6.9, соответствуют определенному сенсору. Каждый содержит данные, отражающие самую последнюю информацию, полученную от соответствующего сенсора. Когда объект данных получает новые данные, он оповещает объекты приемника (ранее зарегистрированные как подлежащие оповещению) в том случае, если изменилось содержимое объекта данных. Любой объект может быть объектом-приемником, если он реализует соответствующий интерфейс. Обычно объекты-приемники отвечают за показ на экране самой последней информации, полученной от сенсора, или за изменение объектов, которые обязаны оповещать другие объекты в том случае, если значения, находящиеся в некоторых объектах данных, удовлетворяют определенным условиям.

Такое взаимодействие требует от объекта `SensorController` изменять содержимое объектов данных. Однако нежелательно, чтобы объекты-приемники изменяли содержимое объектов данных.

При написании объекта-приемника один из способов не дать возможность изменять объект данных состоит в том, чтобы обязать объекты-приемники получать доступ к объектам данных через интерфейс, который не содержит методов, изменяющих содержимое объекта данных. На рис. 6.10 показана соответствующая часть такой организации объектов.

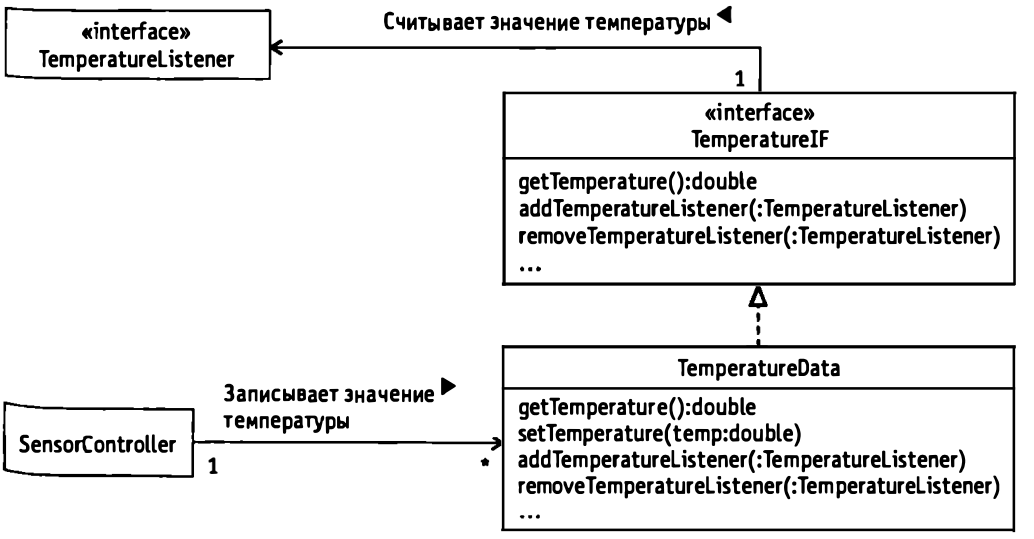


Рис. 6.10. Интерфейс данных

Класс `SensorController` прямым образом использует класс `TemperatureData`. Он имеет возможность задавать значение температуры в объектах `TemperatureData`, вызывая их метод `setTemperature`. Объекты, реализующие интерфейс `TemperatureListener`, могут получать значения температуры от объекта `TemperatureData`, поскольку в интерфейсе `TemperatureIF` объявлен метод `getTemperature`. Однако объекты `TemperatureListener` не могут изменять значение температуры, хранящееся в объекте `TemperatureData`, поскольку они получают доступ к объекту `TemperatureData` через интерфейс `TemperatureIF`, в котором не объявлен метод `setTemperature`.

МОТИВЫ

☺ Существует класс, экземпляры которого одни классы могут изменять, а другие — нет.

☺ Нужно, чтобы экземпляры некоторого класса изменялись бы экземплярами других классов, находящихся в разных пакетах. Это означает объявление методов, которые модифицируют экземпляры классов пакета, закрытыми — не самый пригодный выбор для ограничения количества классов, которые могут вызвать модифицирующие методы.

☹ Необходимо заставить клиентов класса обращаться к этому классу через определяемый самим программистом интерфейс. В целом, если проектируется класс и его клиенты, это правильная политика. Но в некоторых ситуациях это может быть неудобно. Клиентские классы могут разрабатываться третьими фирмами, или просто нежелательно менять интерфейс, используемый клиентскими классами.

☹ Этот шаблон представляет собой хорошую защиту от ошибок программирования, но он не способен воспрепятствовать злонамеренному программированию.

РЕШЕНИЕ

Обеспечиваем доступ только для чтения к изменяемым объектам, требуя от объектов осуществлять доступ к изменяемому объекту через интерфейс, который не содержит каких-либо методов, изменяющих этот объект (рис. 6.11).

Опишем роли, исполняемые классами и интерфейсом в шаблоне Read-Only Interface.

Mutable. Класс, выступающий в этой роли, имеет методы для считывания и записи значений его атрибутов. Он также реализует интерфейс `ReadOnlyIF`.

ReadOnlyIF. Интерфейс, играющий эту роль, имеет такие же методы `get`, что и класс `Mutable`, реализующий этот интерфейс. Однако этот интерфейс не содержит каких-либо методов, которые могут заставить объект `Mutable` изменить свое содержимое.

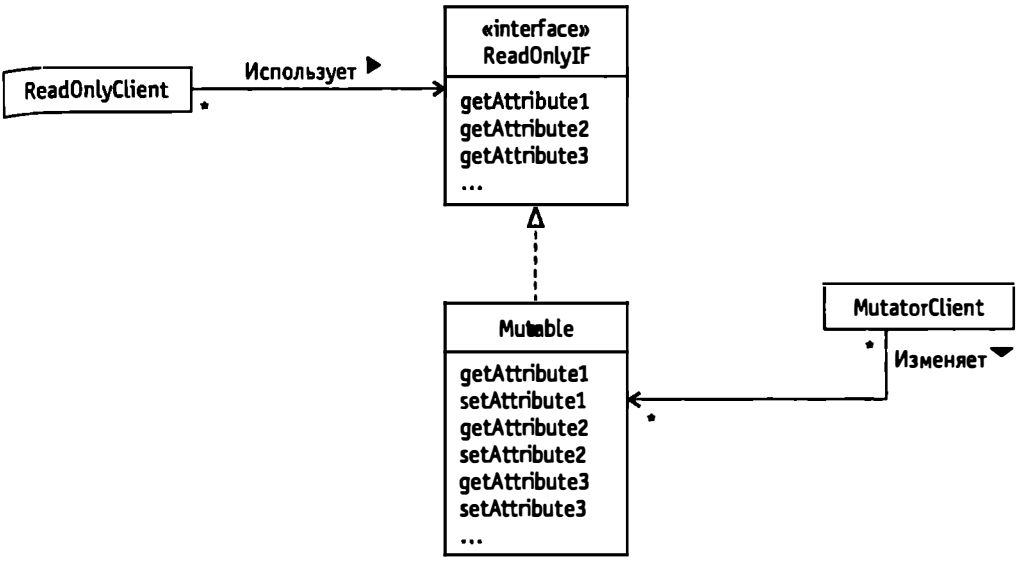


Рис. 6.11. Интерфейс, предназначенный только для чтения

MutatorClient. Классы, выступающие в этой роли, явно используют класс `Mutable` и могут вызывать его методы, изменяющие состояние объекта `Mutable`.

ReadOnlyClient. Классы, играющие эту роль, имеют доступ к классу `Mutable` через интерфейс `ReadOnlyIF`. Классы, которые обращаются к классу `Mutable` через интерфейс `ReadOnlyIF`, не имеют доступа к методам, изменяющим объекты `Mutable`.

РЕАЛИЗАЦИЯ

Если класс `Mutable` должен использоваться как JavaBean, то необходимо учесть некоторые соображения, связанные с представлением изменяемого объекта, предполагающим только его чтение.

Если класс используется как JavaBean, то тот факт, что он реализует интерфейс `ReadOnlyIF`, не помешает программистам изменить состояние экземпляров класса. Это объясняется тем, что при определении, какой из их методов может быть вызван, JavaBeans не зависят от интерфейсов, которые реализует класс. Для этой цели они используют механизм под названием интроспекции (introspection).

Чтобы определить, какие методы бина могут быть вызваны, интроспекция полагается на BeanInfo, связанную с классом бина. Если нет BeanInfo, связанной с этим классом, интроспекция создает BeanInfo по умолчанию. Такая BeanInfo дает программисту доступ к каждому открытому методу, который находит интроспекция.

Чтобы программист не имел доступа к открытым методам, которые изменяют бин, необходимо снабдить его класс явной BeanInfo. BeanInfo должна описывать все свойства бина как предназначенные только для чтения. Кроме того, она не должна обеспечивать какого-либо прямого доступа к тем методам, которые изменяют содержимое бина.

СЛЕДСТВИЯ

☺ Использование шаблона Read-Only Interface защищает от ошибок программирования, которые заключаются в том, что некоторые классы могут изменять те объекты, которые не должны изменять.

☹ Шаблон Read-Only Interface не защищает объекты от ошибочного изменения в результате злонамеренного программирования. Чтобы защититься от него, используйте шаблон Protection Proxy, описанный в книге [Grand2001].

ПРИМЕР КОДА

Пример кода для этого шаблона реализует часть проекта, представленного в разделе «Контекст». Листинг класса `TemperatureData`:

```
public class TemperatureData implements TemperatureIF {
   private double temperature;
   private ArrayList listeners = new ArrayList();

   /**
    * Задает значение температуры, хранящееся в этом объекте.
    */
   public void setTemperature(double temperature) {
      this.temperature = temperature;
      fireTemperature();
   } // setTemperature

   /**
    * Возвращает прочитанное значение температуры,
    * инкапсулированное в этом объекте.
    */
   public double getTemperature() {
      return temperature;
   } // getTemperature()

   /**
    * Добавляем приемник в этот объект.
    */
```

```
    public void addListener(TemperatureIF listener) {
       listeners.add(listener);
    } // addListener(TemperatureIF)

    /**
     * Удаляем приемник из этого объекта.
     */
    public void removeListener(TemperatureIF listener) {
       listeners.remove(listener);
    } // removeListener(TemperatureIF)

    /**
     * Отправляем TemperatureEvent всем
     * зарегистрированным объектам TemperatureListener.
     */
    private void fireTemperature() {
       int count = listeners.size();
       TemperatureEvent evt;
       evt = new TemperatureEvent(this, temperature);
       for (int i = 0; i<count; i++) {
          TemperatureListener thisListener
             = (TemperatureListener)listeners.get(i);
          thisListener.temperatureChanged(evt) ;
       } // for
    } // fireTemperature()

    ...
} // class TemperatureData
```

Объекты `SensorController` записывают значение температуры в объект `TemperatureData`, вызывая его метод `setTemperature`. Считается, что никакой другой класс не может обращаться к методу `setTemperature`. С этой целью другие классы осуществляют доступ к объектам `TemperatureData` через интерфейс `TemperatureIF`:

```
public interface TemperatureIF {
   /**
    * Возвращает прочитанное значение температуры,
    * инкапсулированное в этом объекте.
    */
   public double getTemperature() ;
```

```
    /**
     * Добавляем приемник в этот объект.
     */
    public void addListener(TemperatureIF listener) ;

    /**
     * Удаляем приемник из этого объекта.
     */
    public void removeListener(TemperatureIF listener) ;

    ...
} // interface TemperatureIF
```

Осталось отметить, что интерфейс `TemperatureIF` не имеет метода `setTemperature`.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ READ-ONLY INTERFACE

Interface. Шаблон Read-Only Interface использует шаблон Interface.

Protection Proxy. Шаблон Read-Only Interface не защищает от злонамеренного программирования. Шаблон Protection Proxy, описанный в книге [Grand2001], может использоваться для обеспечения доступа только для чтения к изменяемому объекту в случае злонамеренного программирования. Правда, достигается это за счет дополнительных издержек ввиду сложности программирования и увеличения времени выполнения.

ГЛАВА 7

Структурные шаблоны проектирования

Шаблоны, представленные в этой главе, описывают распространенные способы организации объектов различных типов для совместной работы.

Шаблон Adapter описывает, как у объекта может быть клиент, который предполагает реализацию определенного интерфейса, даже если клиент не реализует этот интерфейс.

Шаблон Iterator показывает, каким образом объект может осуществлять доступ к содержимому коллекции объектов, ничего не зная о структуре или классе этой коллекции.

Шаблон Bridge демонстрирует, как можно управлять параллельными иерархиями абстракций и реализаций.

Шаблон Facade описывает, каким образом можно сокрыть в одном объекте всю сложность, связанную с использованием группы связанных объектов.

Шаблон Flyweight показывает, как можно избежать больших расходов памяти на многочисленные экземпляры некоторого объекта благодаря совместному использованию экземпляров, содержащих общие значения.

Шаблон Dynamic Linkage демонстрирует способ динамического добавления классов в программу на стадии выполнения.

Шаблон Virtual Proxy позволяет отложить на время создание объектов таким способом, который прозрачен для клиентов этих объектов.

Шаблон Decorator предназначен для динамического расширения или изменения поведения существующих объектов.

Шаблон Cache Management показывает, как можно избежать неоднократного создания похожих объектов посредством многократного использования ранее созданного объекта.

Adapter (Адаптер)

Этот шаблон был ранее описан в работе [GoF95].

СИНОПСИС

Класс-адаптер реализует интерфейс, известный его клиентам, и обеспечивает доступ к экземпляру класса, неизвестного его клиентам. Объект-адаптер предоставляет обещанную интерфейсом функциональность, скрывая от клиента информацию о классе, используемом для реализации этого интерфейса.

КОНТЕКСТ

Допустим, создается метод, который копирует массив объектов. Предполагается, что этот метод должен отфильтровывать объекты, не удовлетворяющие определенному критерию, поэтому в копии массива будут содержаться не все элементы, находящиеся в исходном массиве. Для поддержки многократного использования желательно сделать метод не зависящим от конкретного критерия фильтрации. Этого можно достичь, если задать интерфейс, объявляющий метод, который вызывается копировщиком массива с целью выяснения, будет ли данный объект включен в новый массив (рис. 7.1).

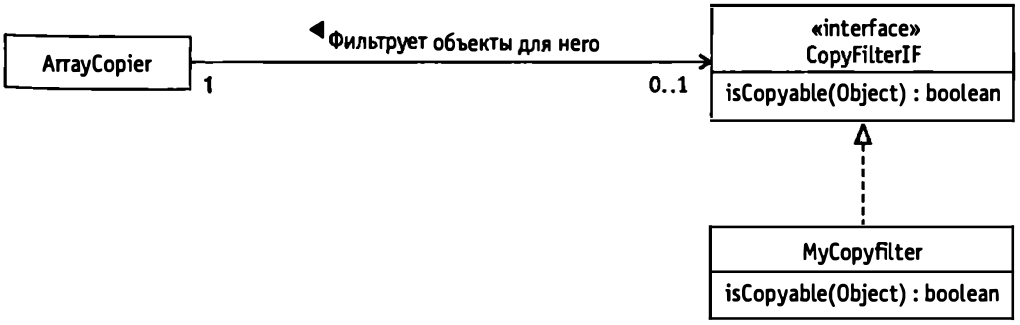


Рис. 7.1. Простой фильтр копирования

Класс `ArrayCopier` делегирует интерфейсу `CopyFilterIF` операцию принятия решения, копировать ли в новый массив элемент старого массива. Если метод `isCopyable` возвращает для объекта true, то этот объект копируется в новый массив.

Данный подход решает проблему, связанную с разрешением инкапсуляции в отдельном объекте критерия копирования, используемого объектами `ArrayCopier`.

не заботясь о том, что представляют собой объекты копирования. Это решение связано с другой проблемой. Иногда необходимая для фильтрации логика находится в методе фильтруемых объектов. Если такие объекты не реализуют интерфейс `CopyFilterIF`, то `ArrayCopier` не может прямо спросить у этих объектов, должны ли они быть скопированы. Однако `ArrayCopier` может косвенно спросить у фильтруемых объектов, должны ли они копироваться, даже если они не реализуют интерфейс `CopyFilterIF`.

Предположим, существует класс `Document`, имеющий метод `isValid`, который возвращает результат типа boolean. Нужно, чтобы объект `ArrayCopier` использовал результат обращения к методу `isValid` с целью выполнения фильтрации во время операции копирования. Класс `Document` не реализует интерфейс `CopyFilterIF`, поэтому объект `ArrayCopier` не может непосредственно использовать объект `Document` для фильтрации. Другой класс, который реализует интерфейс `CopyFilterIF`, не может самостоятельно определить, должен ли объект `Document` копироваться в новый массив. Это объясняется тем, что у него нет способа получить необходимую информацию, не вызывая метод `isValid` объекта `Document`. Решение задачи для этого объекта состоит в том, чтобы вызвать метод `isValid` объекта `Document` и получить результат (рис. 7.2).

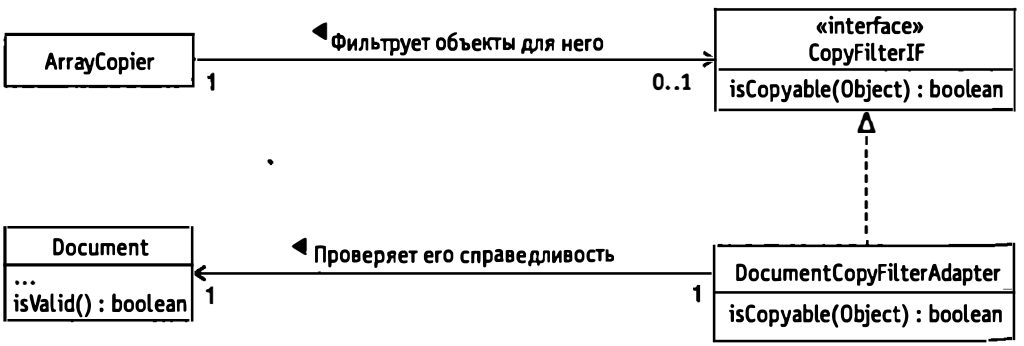


Рис. 7.2. Адаптер фильтра копирования

Согласно этому решению объект `ArrayCopier`, как всегда, вызывает метод `isCopyable` объекта, реализующего интерфейс `CopyFilterIF`. В таком случае этот объект является экземпляром класса `DocumentCopyFilterAdapter`, который реализует метод `isCopyable`, делегируя вызов методу `isValid` объекта `Document`.

МОТИВЫ

☺ Нужно использовать класс, который вызывает метод через интерфейс, причем необходимо использовать его вместе с классом, который не реализует этот интерфейс. Изменение класса таким образом, чтобы он реализовывал интерфейс, неприемлемо по двум причинам.

1. Отсутствует исходный код для класса.
2. Класс является классом общего назначения и не подходит для реализации интерфейса специального назначения.

☺ Нужно динамически определять, какие методы другого объекта вызывает этот объект, причем объект ничего не должен знать о классе другого объекта.

РЕШЕНИЕ

Предположим, есть класс, который вызывает метод через интерфейс. Нужно, чтобы экземпляр этого класса вызывал метод объекта, не реализующего такой интерфейс. Можно сделать так, что экземпляр класса будет производить вызов через объект-адаптер, который реализует интерфейс, делегируя вызовы методу объекта, не реализующего интерфейс (рис. 7.3).

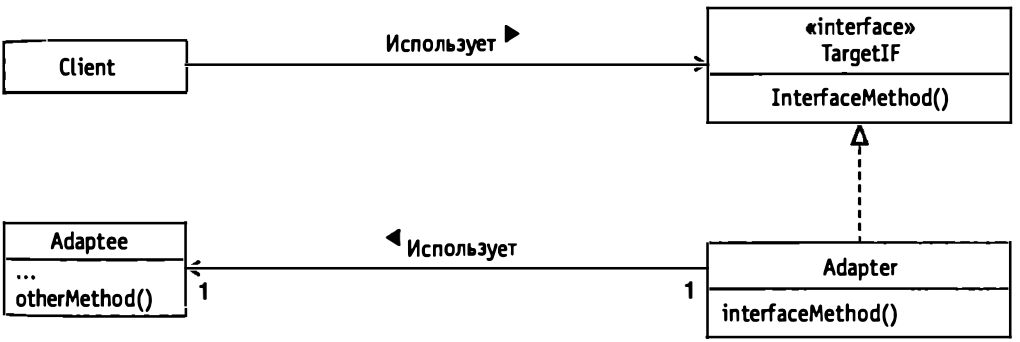


Рис. 7.3. Адаптер

Опишем роли, исполняемые этими классами и интерфейсом.

Client. Класс, выступающий в этой роли, вызывает метод другого класса через интерфейс, чтобы ничего не знать о реальном классе, реализующем этот метод.

TargetIF. Данный интерфейс в этой роли объявляет метод, который вызывается классом `Client`.

Adapter. Этот класс реализует интерфейс `TargetIF`. Он реализует метод, вызываемый клиентом, делегируя вызов методу класса `Adaptee`, не реализующего интерфейс `TargetIF`.

Adaptee. Класс, играющий эту роль, не реализует метод интерфейса `TargetIF`, но содержит метод, который хочет вызывать класс `Client`.

Класс `Adapter` может не только просто делегировать вызов метода. Он может выполнять некоторые преобразования аргументов, а также предоставлять дополнительную логику для сокрытия различий между подразумеваемой семантикой метода интерфейса и реальной семантикой метода класса `Adaptee`.

Сложность класса-адаптера ничем не ограничена. До тех пор, пока основным назначением класса является использование его в качестве промежуточного звена при вызове методов другого объекта, его можно рассматривать как класс-адаптер.

РЕАЛИЗАЦИЯ

Реализуется класс-адаптер очень просто. Однако при реализации шаблона Adapter один вопрос требует внимательного рассмотрения: откуда объекты-адаптеры узнают, какой экземпляр класса `Adaptee` вызывать. Здесь существуют два подхода.

1. Передать ссылку на объект `Adaptee` в качестве параметра для конструктора объекта-адаптера или для одного из его методов. Это позволяет использовать объект-адаптер с любым экземпляром или с любым возможным количеством экземпляров класса `Adaptee`. Такой подход проиллюстрирован в следующем примере:

```
class CustomerBillToAdapter implements AddressIF {
   private Customer myCustomer;

   public CustomerBillToAdapter(Customer customer) {
      myCustomer = customer;
   } // constructor

   public String getAddress1() {
      return myCustomer. getBillToAddress1();
   }
   public void setAddress1(String address1) {
      myCustomer.setBillToAddress1(address1);
   } // setAddress1(String)
...
} // class CustomerBillToAdapter
```

2. Сделать класс-адаптер внутренним классом класса `Adaptee`. При этом упрощается ассоциация между объектом-адаптером и адаптируемым объектом. Кроме того, ассоциация при этом становится жесткой. Данный подход показан в следующем примере:

```
MenuItem exit = new MenuItem(caption);
exit.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent evt) {
          close();
     } // actionPerformed(ActionEvent)
  } );
```

Внутренние классы

Язык Java позволяет использовать в программах вложенные объявления классов, например,

```
public class Foo {
   private int x;
   ...
   class Bar {
      void increment() {
         x = x+1;
      } // increment()
      ...
   } // class bar
} // class foo
```

Поскольку класс `Bar` задан внутри класса `Foo`, он считается частью класса `Foo` и поэтому может ссылаться на закрытые переменные экземпляры класса `Foo`. Когда экземпляр класса `Foo` создает экземпляр класса `Bar`, экземпляр класса `Foo`, который создал объект `Bar`, вызывается включающим его экземпляром класса `Bar`. Любые ссылки в классе `Bar` на одну из переменных экземпляра класса `Foo` будут указывать на переменные включающего его экземпляра.

Внутренние классы могут быть закрытыми. Если некоторый класс нужен только для поддержки другого класса, то самый лучший способ организации поддерживающего класса может состоять в том, чтобы сделать его закрытым внутренним классом того класса, который он поддерживает.

Если объект должен иметь другой объект, вызывающий один из его методов, чтобы сделать это, он передает объект-адаптер другому объекту; в таком случае объект-адаптер вполне может быть экземпляром закрытого внутреннего класса.

Внутренний класс может быть объявлен внутри метода, тогда он может получить доступ к переменным экземпляра содержащего его класса и локальным переменным содержащего его метода.

Внутренние классы, объявленные в методе, могут быть анонимными. Примером такого класса может служить класс, описанный в разделе «Реализация»:

```
MenuItem exit = new MenuItem(caption);
exit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
```

```
            close();
        } // actionPerformed(ActionEvent)
    } );
```

Синтаксис языка одинаков как при создании экземпляра класса, так и при определении класса. Синтаксис предусматривает использование слова «new», за которым следует имя расширяемого класса или реализуемого интерфейса, аргументы конструктора и, наконец, тело класса в фигурных скобках.

Представленный пример определяет анонимный внутренний класс, который реализует интерфейс `ActionListener`. Он создает экземпляр анонимного класса и передает его методу `addActionListener` объекта `MenuItem`.

Более подробная информация о внутренних классах содержится в п. 8.1.2 «Спецификации языка Java» на сайте http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html.

ЛЕДСТВИЯ

Классы `Adaptee` и `Client` остаются независимыми друг от друга.

Класс-адаптер можно использовать для определения, какой из методов объекта вызывается другим объектом. Предположим, есть класс, экземпляры которого представляют собой элементы управления окном GUI, позволяющие отображать на экране и редактировать телефонные номера. Этот класс считывает и сохраняет телефонные номера, вызывая методы, определяемые интерфейсом. Чтобы использовать интерфейс, необходимо определить классы-адаптеры. Один класс-адаптер можно использовать для считывания и хранения номера факса из экземпляров класса, а другой класс — для считывания и хранения номеров пейджера из экземпляров того же класса. Различие между двумя классами адаптеров заключается в том, что они вызывают различные методы класса `Adaptee`. Рассмотрим на примере.

Предположим, что существует класс `PhoneNumberEditor`, который отвечает за то, чтобы разрешить пользователю изменять телефонный номер. Объект передается конструктору класса `PhoneNumberEditor`, который реализует этот интерфейс.

```
public interface PhoneNumberIF {
    public String getPhoneNumber() ;

    public void setPhoneNumber(String newValue) ;
} // interface PhoneNumberIF
```

Если нужно создать два объекта PhoneNumberEditor для изменения офисных телефонных и факсовых номеров какого-то человека, то можно написать примерно такой код:

```
PhoneNumberEditor voiceNumber;
   voiceNumber = new PhoneNumberEditor (new PhoneNumberIF() {
      public String getPhoneNumber() {
         return person.getOfficeNumber();
      } // getPhoneNumber
      public void setPhoneNumber(String newValue) {
         person.setOfficeNumber(newValue);
      } // setPhoneNumber(String)
   }) ;

   PhoneNumberEditor faxNumber;
   faxNumber = new PhoneNumberEditor(new PhoneNumberIF() {
      public String getPhoneNumber() {
         return person.getFAXNumber();
      } // getPhoneNumber

      public void setPhoneNumber(String newValue) {
         person.setFAXNumber(newValue);
      } // setPhoneNumber(String)
   }) ;
```

Каждый объект PhoneNumberEditor создается с новым адаптером. Каждый адаптер вызывает разные методы одного и того же объекта.

☹ Шаблон Adapter добавляет в программу косвенность. Подобно любой другой косвенности, это усложняет понимание такой программы.

ПРИМЕНЕНИЕ В JAVA API И ПРИМЕР КОДА

Распространенный способ использования классов-адаптеров в Java API, предназначенный для обработки события, выглядит примерно так:

```
Button ok = new Button("OK");
ok.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
      doIt();
   } // actionPerformed(ActionEvent)
   } );
add(ok);
```

В этом примере кода создается экземпляр анонимного класса, реализующий интерфейс `ActionListener`. При нажатии кнопки (объект `Button`) вызывается метод `actionPerformed` этого класса. Этот шаблон широко распространен в кодах, предназначенных для обработки событий.

Java API не содержит каких-либо открытых классов-адаптеров, готовых к использованию. Он имеет классы, например `java.awt.event.WindowAdapter`, предназначенные не для прямого использования, а для создания на их основе подклассов. Идея состоит в том, что некоторые интерфейсы приемника событий, например `WindowListener`, объявляют множество методов. Как правило, не все эти методы должны быть реализованы. Интерфейс `WindowListener` объявляет восемь методов, которые вызываются для оповещения о различных событиях, связанных с окнами. Часто только события одного или двух видов представляют интерес. Методы, соответствующие событиям, не представляющим интерес, обычно делают пустыми. Класс `WindowAdapter` реализует интерфейс `WindowListener` и реализует все восемь его методов как бездействующие. Класс адаптера, являющийся подклассом класса `WindowAdapter`, должен реализовывать только методы, соответствующие представляющим интерес событиям. Для всех остальных методов он наследует бездействующие варианты реализации. Например:

```
addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      System.exit();
   } // windowClosing(WindowEvent)
   } );
```

В данном примере кода анонимный класс-адаптер является подклассом класса `WindowAdapter`. Он реализует только метод `windowClosing` и наследует от класса `WindowAdapter` бездействующие варианты реализации остальных семи методов.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ ADAPTER

Facade. Класс `Adapter` предоставляет объект, действующий как промежуточное звено при обращениях к методам, осуществляемых между клиентскими объектами и одним другим объектом, который не известен клиентским объектам. Шаблон Facade предоставляет объект, действующий как промежуточное звено при обращениях к методам, осуществляемым между клиентскими объектами и несколькими объектами, не известными клиентским объектам.

Iterator. Шаблон Iterator представляет собой специальную версию шаблона Adapter, предназначенную для последовательного доступа к содержимому коллекции объектов.

Proxy. Шаблон Proxy, подобно шаблону Adapter, использует объект, который является заменителем другого объекта. Однако объект `Proxy` имеет тот же интерфейс, что и объект, заменителем которого он является.

Strategy. С точки зрения структуры шаблон Strategy аналогичен шаблону Adapter. Различие состоит в предназначении. Шаблон Adapter позволяет объекту `Client` выполнять при взаимодействии свою изначально предопределенную функцию, вызывая методы объектов, реализующих определенный интерфейс. Шаблон Strategy предоставляет объекты, реализующие определенный интерфейс, с целью изменения или определения поведения объекта `Client`.

Anonymous Adapter. Шаблон Anonymous Adapter (описанный в книге [Grand99]) представляет собой шаблон кодирования, который использует анонимные объекты-адаптеры для управления событиями.

Iterator (Итератор)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Iterator определяет интерфейс, который объявляет методы для последовательного доступа к объектам коллекции. Класс, осуществляющий доступ к коллекции только через этот интерфейс, не зависит от класса, реализующего этот интерфейс, и от класса коллекции.

КОНТЕКСТ

Предположим, что создаются классы, которые позволяют просматривать инвентарь на товарном складе. Здесь должен применяться пользовательский интерфейс, который даст возможность пользователю просматривать описание, количество, местонахождение и другую информацию о каждой инвентарной единице.

Классы просмотра инвентаря должны быть частью специализированного приложения. Поэтому они не должны зависеть от реального класса, предоставляющего коллекции предметов инвентаризации. Чтобы обеспечить такую независимость, проектируется интерфейс, позволяющий пользовательскому интерфейсу последовательно обращаться к коллекции инвентарных единиц, ничего не зная об использующемся реальном классе коллекции (рис. 7.4).

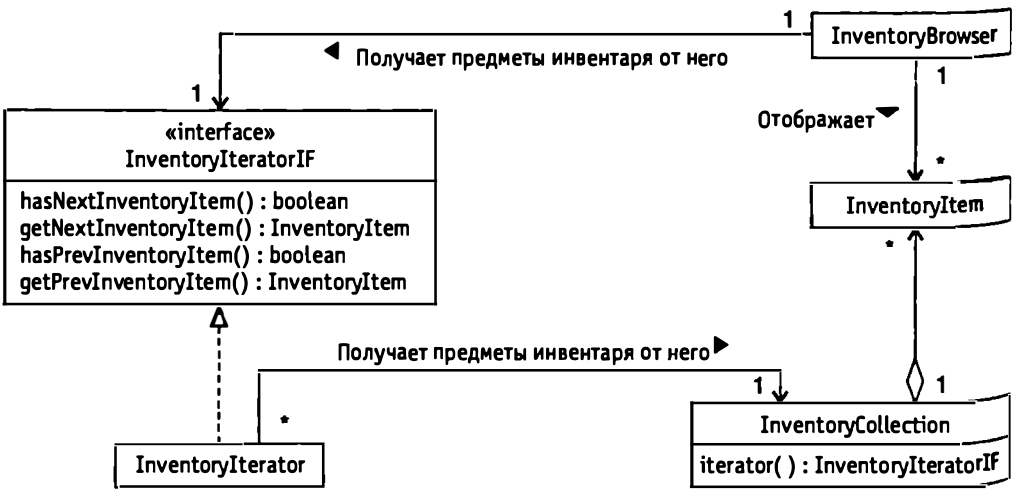


Рис. 7.4. Итератор инвентаря

На этой диаграмме классы пользовательского интерфейса, составляющие просмотрщик инвентаря, описаны как составной класс `InventoryBrowser`. К экземпляру класса `InventoryBrowser` поступает запрос на отображение объектов `InventoryItem`, находящихся в коллекции, инкапсулированной в объекте `InventoryCollection`. Объект `InventoryBrowser` обращается к объекту `InventoryCollection` не прямым образом. Вместо этого ему предоставляется объект, который реализует интерфейс `InventoryIteratorIF`. Интерфейс `InventoryIteratorIF` определяет методы, позволяющие объекту последовательно считывать содержимое коллекции объектов `InventoryItem`.

МОТИВЫ

☺ Класс нуждается в доступе к содержимому коллекции, не становясь зависимым от класса, который используется для реализации коллекции.

☺ Классу нужен универсальный способ доступа к содержимому множества коллекций.

РЕШЕНИЕ

Диаграмма классов, представленная на рис. 7.5, описывает организацию классов и интерфейсов, участвующих в шаблоне Iterator.

Рассмотрим роли, исполняемые этими классами и интерфейсами.

Collection. Класс в этой роли инкапсулирует коллекцию объектов или значений.

IteratorIF. Интерфейс в этой роли определяет методы для последовательного доступа к объектам, которые инкапсулированы в объекте `Collection`.

Iterator. Класс в этой роли реализует интерфейс `IteratorIF`. Его экземпляры обеспечивают последовательный доступ к содержимому объекта `Collection`, связанного с объектом `Iterator`.

CollectionIF. Обычно классы `Collection` берут на себя ответственность за создание собственных объектов-итераторов. Очень удобно иметь универсальный способ, позволяющий запрашивать объект `Collection` с целью создания объекта-итератора (`Iterator`) для самого себя. Чтобы обеспечить эту универсальность, все классы `Collection` реализуют интерфейс `CollectionIF`, который объявляет метод, предназначенный для создания объектов `Iterator`.

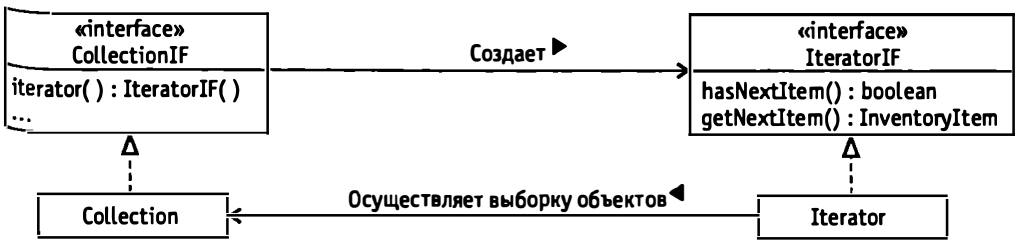


Рис. 7.5. Шаблон Iterator

РЕАЛИЗАЦИЯ

Дополнительные методы

Интерфейс объекта-итератора, представленный в разделе «Решение», содержит минимальный набор методов. Интерфейсы объектов-итераторов, как правило, определяют дополнительные методы, если они нужны и если поддерживаются базовой коллекцией классов. Кроме методов, проверяющих наличие и считывающих следующий элемент коллекции, часто используются следующие методы:

- проверка наличия и считывание предыдущего элемента коллекции;
- перемещение к первому или последнему элементу коллекции;
- получение количества элементов обхода.

Внутренний класс

Во многих случаях алгоритм обхода класса-итератора требует доступа к внутренней структуре данных класса коллекции. По этой причине классы-итераторы часто реализуются в виде закрытого внутреннего класса, принадлежащего классу коллекции.

Пустой итератор

Пустой итератор — это итератор, который не возвращает никаких объектов. Его метод `hasNext` всегда возвращает false. Пустые итераторы обычно представляют собой простой класс, который реализует соответствующий интерфейс итератора. Использование пустых итераторов может упростить реализацию классов коллекции и других классов итераторов, так как в этом случае нет необходимости в коде, предназначенном для обработки специального случая нулевого обхода.

Изменение базовой коллекции

При изменении коллекции во время обхода итератором ее содержимого могут возникнуть проблемы. Если при выполнении таких изменений не принять мер предосторожности, то итератор может возвратить несогласованный набор результатов. Возможны следующие ошибки: пропуск объектов или возврат одного и того же объекта дважды.

Самый простой способ управления изменениями базовой коллекции во время обхода итератором заключается в том, чтобы после изменения базовой коллекции считать итератор неправильным. Чтобы это реализовать, каждый класс коллекции должен иметь методы, которые увеличивают счетчик в случае ее изменения. Объекты-итераторы могут обнаружить изменение своей базовой коллекции, фиксируя разные показания счетчика изменений. Если метод объекта-итератора извещается, что базовая коллекция была изменена, то он может сгенерировать исключение.

Более надежный способ управления изменениями базовой коллекции во время обхода итератора заключается в том, чтобы гарантировать возврат итератором согласованного набора результатов. Для решения этой задачи применяется несколько способов. Хотя выполнение полного копирования базовой коллекции дает хорошие результаты в большинстве случаев, это, как правило, самый нежелательный подход, поскольку он требует максимальных затрат времени и памяти.

СЛЕДСТВИЯ

- ☺ Доступ к коллекции объектов возможен при отсутствии сведений об источнике объектов.
- ☺ При использовании множества объектов-итераторов очень просто осуществлять и управлять несколькими обходами одновременно.
- ☺ Класс коллекции может предоставлять различные объекты итерации, которые обходят коллекцию по-разному. Например, класс коллекции, поддерживающий ассоциацию между объектами ключей и объектами значений, может иметь одни методы создания итераторов, которые обходят только объекты ключей, и другие методы для создания итераторов, которые обходят только объекты значений.

ПРИМЕНЕНИЕ В JAVA API

Классы коллекций в пакете `java.util` созданы в соответствии с шаблоном Iterator. Интерфейс `java.util.Collection` играет роль `CollectionIF`. Пакет содержит ряд классов, реализующих `java.util.Collection`.

Интерфейс `java.util.Iterator` выполняет роль `IteratorIF`. Классы этого пакета, реализующие `java.util.Collection`, определяют внутренние закрытые классы, которые реализуют `java.util.Iterator` и играют роль итератора.

ПРИМЕР КОДА

В качестве примера кода рассмотрим некоторый скелет программы, реализующей проект, который описан в разделе «Контекст». Листинг для интерфейса `InventoryIteratorIF`:

```
public interface InventoryIteratorIF {
   public boolean hasNextInventoryItem() ;
   public InventoryItem getNextInventoryItem() ;
   public boolean hasPrevInventoryItem() ;
   public InventoryItem getPrevInventoryItem() ;
} // interface InventoryIterator
```

Приведем скелетный листинг для класса `InventoryCollection`. Листинг содержит метод `iterator`, используемый другими классами для получения объекта, необходимого для обхода содержимого объекта `InventoryCollection`. Он включает в себя также закрытый класс, который инстанциируется методом `iterator`.

```
public class InventoryCollection {
   ...
   public InventoryIteratorIF iterator() {
      return new InventoryIterator();
   } // iterator()

   private class InventoryIterator
         implements InventoryIteratorIF {
      public boolean hasNextInventoryItem() {
         ...
      } // hasNextInventoryItem()

      public InventoryItem getNextInventoryItem() {
         ...
      } // getNextInventoryItem()

      public boolean hasPrevInventoryItem() {
         ...
      } // hasPrevInventoryItem()
      public InventoryItem getPrevInventoryItem() {
         ...
      } // getPrevInventoryItem()
   } // class InventoryIterator
   ...
} // class InventoryCollection
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ ITERATOR

Adapter. Шаблон Iterator — специальная форма шаблона Adapter, предназначенная для последовательного доступа к содержимому объектов коллекции.

Factory Method. Некоторые классы коллекций могут использовать шаблон Factory Method для принятия решения, итератор какого вида должен быть инстанциирован.

Null Object. Пустые итераторы иногда используются для реализации шаблона Null Object.

Bridge (Мост)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Bridge полезен в случаях, когда существует иерархия абстракций и соответствующая иерархия реализаций. Шаблон Bridge не комбинирует абстракции и реализации в виде множества отдельных классов, а реализует их в виде независимых динамически объединяемых классов.

КОНТЕКСТ

Предположим, что нужно написать классы, которые осуществляют доступ к физическим сенсорам, контролирующим приложения. Сенсорами будем считать некоторые устройства, например, шкалы, устройства измерения скорости и датчики определения позиции. Общим для всех этих устройств является то, что они выполняют физическое измерение и выдают значение. Отличаются эти устройства только типом производимых ими измерений.

- Шкала выдает единственное число, являющееся результатом измерения в одной точке в некоторый момент времени.
- Устройство измерения скорости производит единственное измерение, представляющее собой среднее значение за какой-то период.
- Датчик определения позиции выдает поток измерений.

Это означает, что такие устройства могут поддерживаться тремя классами, соответствующими различным способам измерений (рис. 7.6).

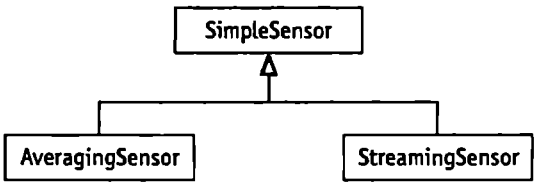


Рис. 7.6. Классы сенсоров

Три данных класса описывают чистые абстракции, которые могут применяться не только к трем вышеназванным, но и к сенсорам многих других типов. Существуют сенсоры других типов, выполняющие простые измерения, измерения с усреднением по времени и потоки измерений, поэтому можно многократно

использовать эти классы для датчиков таких типов. Проблема, связанная с реализацией подобного многократного использования, состоит в том, что детали установления связи с сенсорами, сделанными различными производителями, отличаются друг от друга. Предположим, что создаваемое ПО должно работать с сенсорами двух производителей: Eagle и Hawk. Решение может быть найдено, если каждому производителю будет соответствовать свой класс (рис. 7.7).

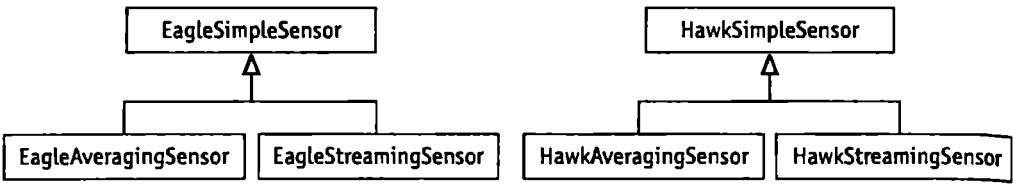


Рис. 7.7. Классы сенсоров, зависящие от производителей

Еще одна проблема заключается не только в невозможности многократного применения классов для простых, усредняющих и потоковых датчиков. При использовании других классов также обнаруживаются различия между производителями, что делает такие классы менее пригодными для многократного использования. Задача состоит в том, как представить иерархию абстракций, чтобы абстракции поддерживались независимо от их реализаций.

С этой целью следует добавить косвенность, которая должна защитить иерархию классов, поддерживающих абстракции, от классов, реализующих эти абстракции. Имеется в виду, что классы абстракций должны иметь доступ к классам реализаций через иерархию интерфейсов реализаций, параллельную иерархии абстракций (рис. 7.8).

Согласно представленному на данном рисунке решению проблема состоит из трех иерархий, которые отделены друг от друга горизонтальными линиями:

- вверху располагается иерархия классов сенсоров, не зависящих от производителя;
- внизу располагаются параллельные иерархии классов, зависящих от производителя;
- в центре располагается параллельная иерархия интерфейсов, которые позволяют классам, не зависящим от производителя, оставаться независимыми от любых классов, специфических для производителя.

Логика, общая для сенсора любого типа, будет находиться в классе, не зависящем от производителя. Логика, отражающая специфику производителя, попадет в класс, который зависит от производителя.

Большая часть рассматриваемой в этом примере логики предназначена для управления исключительными ситуациями. Примером такой ситуации может быть следующее: простой сенсор обнаруживает выходящее за пределы диапазона значение, слишком большое, чтобы его можно было измерить. Не зависящее от

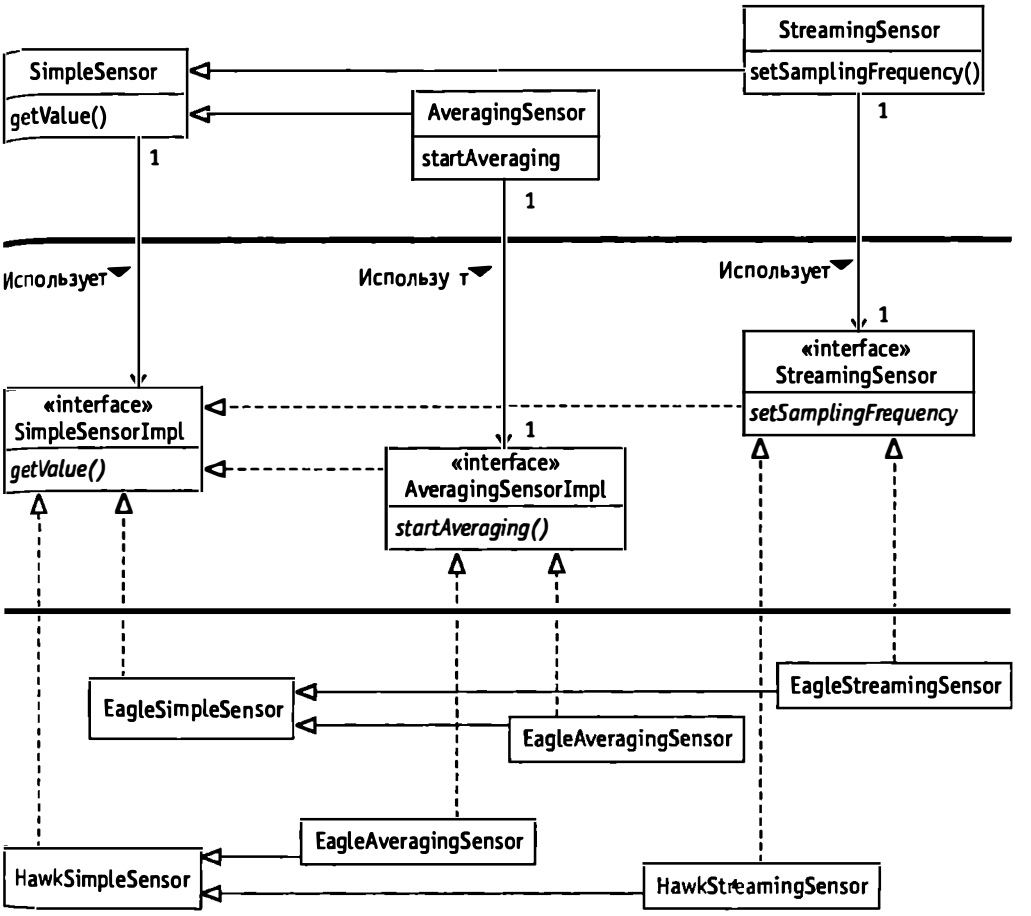


Рис. 7.8. Независимые классы сенсоров и производителей сенсоров

производителя управление этой ситуацией могло бы преобразовать значение и сделать его равным максимальному значению, предварительно установленному для данного приложения. Управление этой ситуацией, зависящее от производителя, может потребовать, чтобы любые считываемые сенсором данные не считались правильными до тех пор, пока не пройдет какое-то определенное время после окончания ситуации выхода за пределы диапазона.

МОТИВЫ

☺ Если иерархии абстракций и иерархии их реализаций объединяются в единую иерархию классов, то классы, использующие такие классы, будут связаны с определенной реализацией абстракции. Изменение реализации соответствующей абстракции не должно приводить к изменению классов, использующих эту абстракцию.

- ☺ Нужно многократно использовать логику, общую для различных реализаций абстракции. Обычный способ сделать логику многократно используемой состоит в том, чтобы инкапсулировать ее в отдельном классе.
- ☺ Необходимо иметь возможность создавать новую реализацию абстракции, не выполняя повторную реализацию общей логики абстракции.
- ☺ Нужно расширить общую логику абстракции путем написания одного нового класса, а не путем написания нового класса для каждой комбинации: «базовая абстракция — ее реализации».
- ☺ Если есть возможность, несколько абстракций должны совместно использовать одну и ту же реализацию.

РЕШЕНИЕ

Шаблон Bridge позволяет классам, соответствующим некоторым абстракциям, существовать отдельно от классов, реализующих эти абстракции. Можно поддерживать чистое разделение, если классы абстракций имеют доступ к классам реализации через интерфейсы, которые образуют иерархию, параллельную иерархии наследования классов абстракций (рис. 7.9).

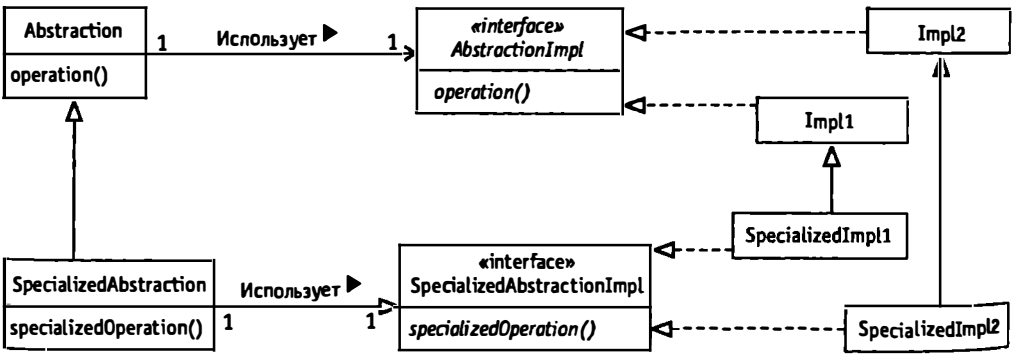


Рис. 7.9. Шаблон Bridge

Опишем роли, исполняемые классами и интерфейсами в шаблоне Bridge.

Abstraction. Этот класс представляет абстракцию верхнего уровня. Он отвечает за поддержание ссылки на объект, который реализует интерфейс `AbstractionImpl`, поэтому может делегировать операции выполнение ее реализации. Если экземпляр класса `Abstraction` является также экземпляром подкласса класса `Abstraction`, то этот экземпляр будет ссылаться на объект, который реализует соответствующий интерфейс-наследник интерфейса `AbstractionImpl`.

SpecializedAbstraction. Эту роль может играть любой подкласс класса `Abstraction`. Для каждого такого подкласса класса `Abstraction` имеется соответст-

вующий интерфейс-наследник интерфейса AbstractionImpl. Каждый класс SpecializedAbstraction делегирует свои операции объекту реализации, который реализует интерфейс SpecializedAbstractionImpl, соответствующий классу SpecializedAbstraction.

AbstractionImpl. Этот интерфейс объявляет методы для всех операций нижнего уровня, которые должны предоставляться реализацией класса Abstraction.

SpecializedAbstractionImpl. Он соответствует интерфейсу-наследнику интерфейса AbstractionImpl. Каждый интерфейс SpecializedAbstractionImpl соответствует классу SpecializedAbstraction и объявляет методы для операций нижнего уровня, необходимых для реализации этого класса.

Impl1, Impl2. Эти классы реализуют интерфейс AbstractionImpl и обеспечивают различные реализации класса Abstraction.

SpecializedImpl1, SpecializedImpl2. Эти классы реализуют один из интерфейсов SpecializedAbstractionImpl и обеспечивают различные реализации класса SpecializedAbstraction.

На рис. 7.9 представлены интерфейсы реализации абстракций, имеющие те же методы, что и соответствующие классы абстракций. Это сделано просто для наглядности. Интерфейсы реализации абстракций могут содержать методы, отличающиеся от методов соответствующих классов абстракций.

РЕАЛИЗАЦИЯ

При реализации шаблона Bridge всегда должна решаться следующая задача: как создавать объекты реализации для каждого объекта абстракции. Наиболее часто встречаются решения, при которых объекты абстракций создают свои собственные объекты реализаций либо делегируют создание этих объектов реализаций другому объекту.

Самым оптимальным обычно является второй вариант, при котором объекты абстракций делегируют создание объектов реализаций. При этом сохраняется взаимная независимость классов абстракций и реализаций. Если классы абстракций должны делегировать создание объектов реализаций, то для создания объектов реализаций в проекте обычно используется шаблон Abstract Factory.

Однако если количество классов реализаций для абстрактного класса очень мало и набор классов реализаций, как предполагается, не должен изменяться, то приемлемая оптимизация заключается в том, чтобы заставить класс абстракции создавать свои собственные объекты реализации.

С этой проблемой связано принятие еще одного решения: будет ли объект абстракции использовать один и тот же объект реализации в течение времени жизни. По мере изменения используемых шаблонов или других условий может быть уместно изменение объекта реализации, используемого объектом абстракции. Если класс абстракции прямым образом создает собственные объекты реализаций, то лучше поместить логику, отвечающую за изменение объекта

реализации, прямо в класс абстракции. В противном случае можно использовать шаблон Decorator с целью инкапсуляции логики, предназначенной для переключения объектов реализации, в классе-обертке (wrapper class).

СЛЕДСТВИЯ

☺ Шаблон Bridge поддерживает классы, представляющие абстракцию, независимо от классов, реализующих ее. Абстракция и ее реализации организованы в виде отдельных иерархий классов. Можно расширить каждую иерархию классов без непосредственного воздействия на другую иерархию классов. Один класс абстракции может иметь множество классов реализаций, или несколько классов абстракций могут использовать один и тот же класс реализации.

☺ Классы, которые являются клиентами классов абстракций, не обладают какой-либо информацией о классах реализаций. Поэтому объект абстракции может изменять свою реализацию, не оказывая никакого влияния на своих клиентов.

ПРИМЕНЕНИЕ В JAVA API

Java API содержит пакет `java.awt`, в котором находится класс `Component`. Он представляет собой абстрактный класс, инкапсулирующий общую для всех компонентов GUI логику. Класс `Component` имеет подклассы, например, `Button`, `List` и `TextField`, которые инкапсулируют логику для соответствующих компонентов GUI, не зависящую от платформы. В пакете `java.awt.peer` есть также интерфейсы, например, `ComponentPeer`, `ButtonPeer`, `ListPeer` и `TextFieldPeer`, которые объявляют методы, необходимые для классов реализаций, обеспечивающих зависящую от платформы поддержку подклассов класса `Component`.

Подклассы класса `Component` используют шаблон Abstract Factory для создания своих объектов реализаций. Класс `java.awt.Toolkit` — это абстрактный класс, исполняющий роль абстрактной фабрики. В зависимости от платформы предоставляются классы реализаций и класс конкретной фабрики, используемый для инстанциирования классов реализаций.

ПРИМЕР КОДА

В качестве примера шаблона Bridge рассмотрим код, реализующий классы, связанные с сенсорами и рассмотренные в разделе «Контекст». Предположим, что объекты, представляющие сенсоры и их реализации, создаются при помощи шаблона Factory Method. Объект `Factory Method` знает, какие сенсоры доступны и какие объекты должны создаваться для предоставления доступа к сенсору, а также создаст такие объекты при первом же запросе на получение доступа к сенсору.

Приведем код для класса SimpleSensor, исполняющего роль класса абстракции:

```
public class SimpleSensor {
   // Ссылка на объект, который реализует операции,
   // специфические для реального сенсорного устройства,
   // представленного этим объектом.
   private SimpleSensorImpl impl;
   /**
    * @param impl
    *     Объект, который реализует операции, зависящие от типа
    *     сенсора.
    */
   SimpleSensor(SimpleSensorImpl impl) {
      this.impl = impl;
   } // constructor (SimpleSensorImpl)

   protected SimpleSensorImpl getImpl() {
      return impl;
   } // getImpl()
...
   /**
    * Возвращает значение, которое является текущим измерением
    * сенсора.
    */
   public int getValue() throws SensorException {
      return impl.getValue();
   } // getValue()
} // class SimpleSensor
```

Как следует из названия[1], класс SimpleSensor является простым. Он делает немного больше, чем просто делегирует свои операции объекту, реализующему интерфейс SimpleSensorImpl. Приведем код интерфейса SimpleSensorImpl:

```
interface SimpleSensorImpl {
   /**
    * Возвращает значение, которое является текущим измерением
    * сенсора.
    */
   public int getValue() throws SensorException;
} // interface SimpleSensorImpl
```

[1] От англ. simple — простой. (*Примеч. ред.*)

Некоторые подклассы класса `SimpleSensor` обладают такой же простой структурой. Приведем код класса `AveragingSensor`. Экземпляры этого класса представляют сенсоры, которые выдают значения, определяющие среднее всех измерений, выполненных в течение некоторого периода времени.

```
public class AveragingSensor extends SimpleSensor {
   /**
    * @param impl
    *     Объект, реализующий операции, зависящие от типа сенсора.
    */
   AveragingSensor(AveragingSensorImpl impl) {
     super(impl);
   } // constructor(AveragingSensorImpl)
...
   /**
    * Усредняющие сенсоры выдают значение, которое является
    * средним всех измерений, выполненных в течение некоторого
    * периода.
    * Этот период начинается с момента вызова этого
    * метода.
    */
   public void beginAverage() throws SensorException {
     ((AveragingSensorImpl) getImpl()).beginAverage();
   } // beginAverage()
} // class AveragingSensor
```

Нетрудно заметить, что класс `AveragingSensor` является очень простым, он делегирует свои операции используемым объектам реализации.

```
interface AveragingSensorImpl extends SimpleSensorImpl {
   /**
    * Усредняющие сенсоры выдают значение, которое является
    * средним всех измерений, выполненных в течение некоторого
    * периода.
    * Этот период начинается с момента вызова этого
    * метода.
    */
   public void beginAverage() throws SensorException;
} // interface AveragingSensorImpl
```

Подклассы класса SimpleSensor могут быть более сложными и предоставлять собственные дополнительные сервисы. Класс StreamingSensor передает поток измерений объектам, зарегистрировавшимся в качестве получателей этих измерений. Он передает измерение методу того объекта, который должен получать измерение. Он никак не ограничивает время, которое потребуется на выполнение метода. Здесь существует простое предположение, что время выполнения метода будет вполне приемлемым. С другой стороны, объекты реализации, используемые вместе с экземплярами класса StreamingSensor, могут нуждаться в предоставлении измерений с постоянной скоростью, или же данные будут потеряны. Чтобы предотвратить потерю измерений, экземпляры класса StreamingSensor помещают предоставляемые им данные измерений в буфер, откуда эти данные асинхронно передаются другим объектам. Теперь рассмотрим код для класса StreamingSensor:

```
public class StreamingSensor extends SimpleSensor
            implements StreamingSensorListener, Runnable {
  // Эти объекты предоставляют буфер, позволяющий объекту
  // реализации асинхронно передавать измеренные значения в то
  // время, когда этот объект передает  уже полученное значение
  // своим приемникам.
  private DataInputStream consumer;
  private DataOutputStream producer;

  // Коллекция приемников.
  private Vector listeners = new Vector();

  /**
   * @param impl
   *     Объект, который реализует операции, зависящие от типа
   *     сенсора и предоставляемые этим объектом.
   */
  StreamingSensor(StreamingSensorImpl impl)
            throws SensorException {
    super(impl);

    // Создает канальный поток, который будет поддерживать
    // способность этого объекта передавать измеренные данные
    // одновременно с их получением.
    PipedInputStream pipedInput = new PipedInputStream();
    consumer = new DataInputStream(pipedInput);
    PipedOutputStream pipedOutput;
    try {
      pipedOutput = new PipedOutputStream(pipedInput);
```

```
        } catch (IOException e) {
            throw new SensorException(«pipe creation failed»);
        } // try
        producer = new DataOutputStream(pipedOutput);

        // Запускаем поток для передачи измеренных значений.
        new Thread(this).start();
    } // constructor(StreamingSensorImpl)
...
    /**
     * Потоковые сенсоры выдают поток измеренных величин.
     * Частота выдачи потока значений не превышает заданное
     * количество раз в минуту.
     * @param freq
     *     Максимальная частота (раз в минуту) выдачи результатов
     *     измерений данным потоковым сенсором.
     */
    public void setSamplingFrequency(int freq)
                throws SensorException {
        // Делегирует это объекту реализации.
        StreamingSensorImpl impl;
        impl = (StreamingSensorImpl)getImpl();
        impl.setSamplingFreguency(freq);
    } // setSamplingFrequency(int)

    /**
     * Объекты StreamingSensor предоставляют поток значений
     * заинтересованным в этом объектам,
     * передавая каждое значение методу processMeasurement
     * объекта. Передача значений осуществляется при помощи
     * собственного потока и  является полностью асинхронной.
     * @param value Передаваемое измеренное  значение.
     */
    public void processMeasurement(int value) {
        try {
            producer.writeInt(value);
        } catch (IOException e) {
        // Не может передать значение, просто отбрасывает его.
        } // try
    } // processMeasurement(int)
```

```
  /**
   * Метод регистрирует получателя будущих
   * данных измерений от этого сенсора.
   */
  public void addStreamingSensorListener(
            StreamingSensorListener listener) {
    listeners.addElement(listener);
  } // addStreamingSensorListener(StreamingSensorListener)

  /**
   * Этот метод отменяет регистрацию получателя
   * будущих данных от этого сенсора.
   */
  public void removeStreamingSensorListener(
            StreamingSensorListener listener) {
    listeners.removeElement(listener);
  } // addStreamingSensorListener(StreamingSensorListener)

  /**
   * Этот метод асинхронно удаляет измеренные значения
   * из канала и передает их зарегистрированным получателям.
   */
  public void run() {
    while (true) {
      int value;
      try {
        value = consumer.readInt();
      } catch (IOException e) {
        // Канал не работает, поэтому выходим из метода,
        // что приводит к окончанию выполнения потока.
        return;
      } // try
      for (int i=0; i < listeners.size(); i++) {
        StreamingSensorListener listener;
        listener
          = (StreamingSensorListener)listeners.elementAt(i);
        listener.processMeasurement(value);
      } // for
    } // while
  } // run()
} // class StreamingSensor
```

Чтобы класс StreamingSensor мог передавать измерение объекту, необходимо, чтобы этот объект реализовывал интерфейс StreamingSensorListener. Измерения передаются методу processMeasurement, который объявлен интерфейсом StreamingSensorListener. Класс StreamingSensor тоже реализует интерфейс StreamingSensorListener. Объекты реализации передают измерения экземплярам класса StreamingSensor, вызывая его метод processMeasurement.

И наконец, приведем интерфейс, который соответствует классу StreamingSensor:

```
interface StreamingSensorImpl extends SimpleSensorImpl {
   /**
    * Потоковые сенсоры выдают поток измеренных значений.
    * Частота выдачи потока значений не превышает
    * заданное количество раз в минуту.
    * @param freq
    *     Максимальная частота (раз в минуту) выдачи результатов
    *     измерений данным потоковым сенсором.
    */
   public void setSamplingFreguency(int freq)
     throws SensorException;

   /**
    * Данный метод вызывается объектом, представляющим абстракцию
    * потокового сенсора таким образом, что этот объект может
    * выполнить обратный вызов к такому объекту для передачи ему
    * измеренных значений.
    * @param abstraction Объект абстракции, которому передаются
    * измеренные значения.
    */
   public void setStreamingSensorListener(
             StreamingSensorListener listener);
} // interface StreamingSensorImpl
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ BRIDGE

Layered Architecture. Шаблон Bridge представляет собой способ организации сущностей (в виде классов), идентифицированных с помощью шаблона Layered Architecture, описанного в работе [BMRSS96].

Abstract Factory. Шаблон Abstract Factory может использоваться шаблоном Bridge для принятия решения о том, какой класс реализации нужно инстанциировать для объекта абстракции.

Decorator. Шаблон Decorator может использоваться для динамического выбора объекта реализации, которому объект абстракции будет делегировать некоторую операцию.

Facade (Фасад)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Facade упрощает доступ к связанному набору объектов, предоставляя для связи с этим набором объектов один объект, который используется всеми другими объектами, не принадлежащими этому набору объектов.

КОНТЕКСТ

Рассмотрим организацию классов, поддерживающих создание и передачу сообщений электронной почты. Сюда могут входить следующие классы:

- класс `MessageBody`, экземпляры которого содержат тела сообщений;
- класс `Attachment`, экземпляры которого содержат прилагаемую к сообщению информацию, которая может быть присоединена к объекту `MessageBody`;
- класс `MessageHeader`, экземпляры которого содержат информацию для заголовка — кому, от кого, тема (to, from, subject) — сообщения электронной почты;
- класс `Message`, экземпляры которого связаны с объектом `MessageHeader` и объектом `MessageBody`;
- класс `Security`, экземпляры которого используются для добавления цифровой подписи в сообщение;
- класс `MessageSender`, экземпляры которого должны отправлять объекты `Message` на сервер, отвечающий за доставку электронной почты по нужному адресу или на другой сервер.

На рис. 7.10 представлена диаграмма классов, описывающая отношения между этими и клиентским классом.

Очевидно, что работа с такими классами электронной почты усложняет клиентский класс. Чтобы применять эти классы, клиент должен иметь информацию как минимум о шести классах, об отношениях между ними и о порядке создания экземпляров таких классов. Если каждый клиент этих классов должен учитывать подобную дополнительную сложность, то многократное использование классов электронной почты весьма затруднено.

Шаблон Facade предоставляет способ защиты клиентов классов, например, классов электронной почты, от всей сложности организации таких классов. Этот способ подразумевает наличие дополнительного объекта, который скры-

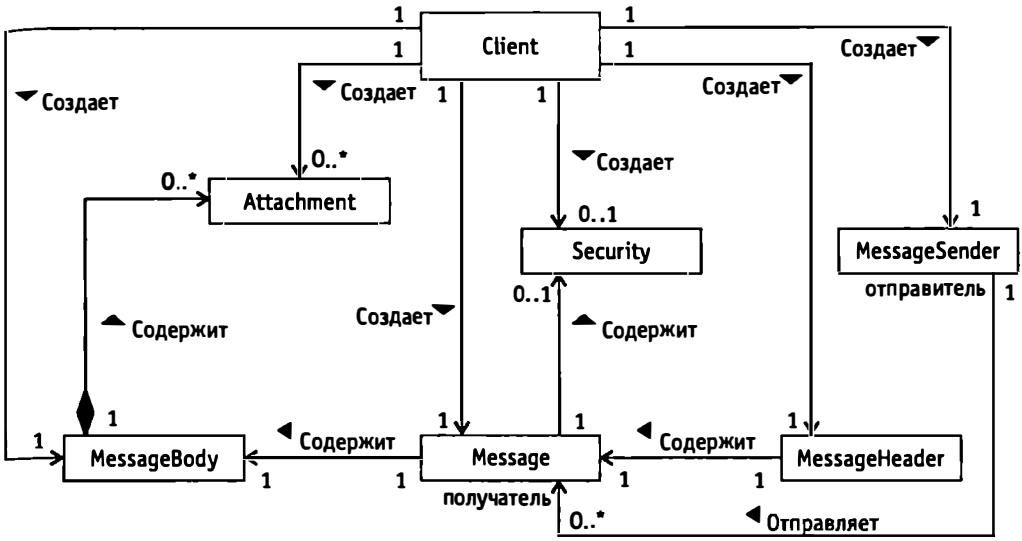


Рис. 7.10. Создание электронного письма

вает от клиентских классов большую часть сложности работы с другими классами. На рис. 7.11 представлена диаграмма классов, демонстрирующая такую организацию, более пригодную для многократного использования.

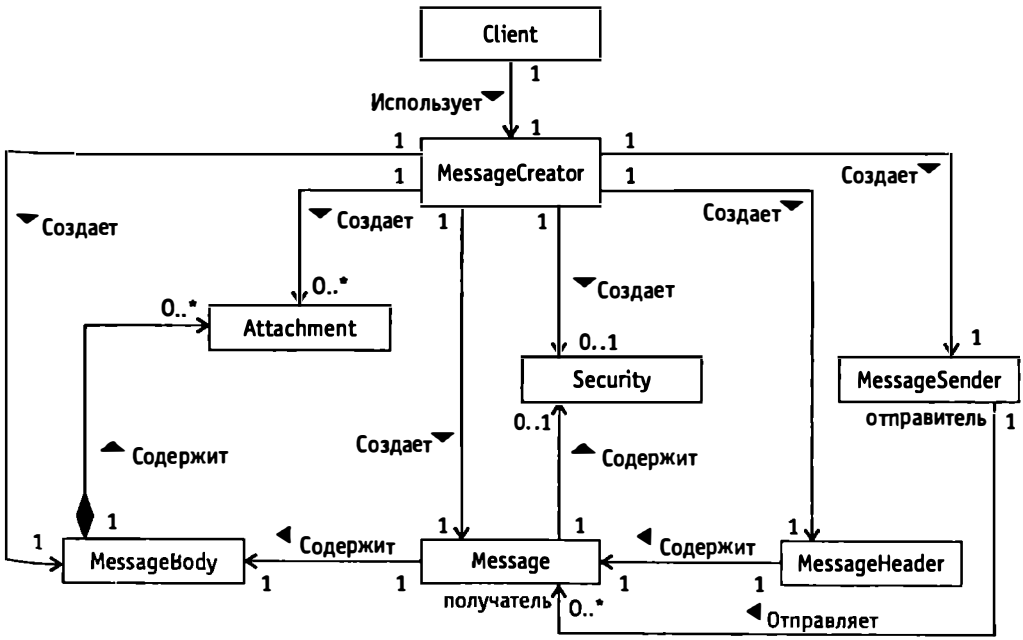


Рис. 7.11. Создание электронного письма с использованием шаблона Facade

На этой схеме та часть класса `Client`, которая отвечала за взаимодействие с классами электронной почты, теперь объединена в один независимый класс, и клиентские классы должны знать только о классе `MessageCreator`. Более того, логика создания всего сообщения, его частей, а также порядка, в котором эти части должны создаваться, сокрыта внутри класса `MessageCreator`, что избавляет клиентские классы от необходимости знать всю эту информацию.

МОТИВЫ

☺ Между классами, реализующими абстракцию, и соответствующими клиентскими классами существует множество зависимостей. Эти зависимости очень усложняют клиентские классы.

☺ Нужно упростить клиентские классы, поскольку более простые классы позволяют делать меньше ошибок. Кроме того, чем проще клиенты, тем требуются меньшие усилия для использования классов, реализующих абстракцию, при их изменении.

☺ Проектируются классы для многоуровневого приложения. Необходимо свести к минимуму количество классов, доступных с соседних уровней.

РЕШЕНИЕ

На рис. 7.12 представлена диаграмма классов, демонстрирующая основную структуру шаблона Facade. Объект клиента взаимодействует с объектом `Facade`, который обеспечивает необходимую функциональность, взаимодействуя с остальными объектами. Если лишь некоторые клиенты нуждаются в дополнительной функциональности, то вместо непосредственного предоставления этой функциональности объект `Facade` предоставляет метод для доступа к другому объекту, который и обеспечивает эту функциональность.

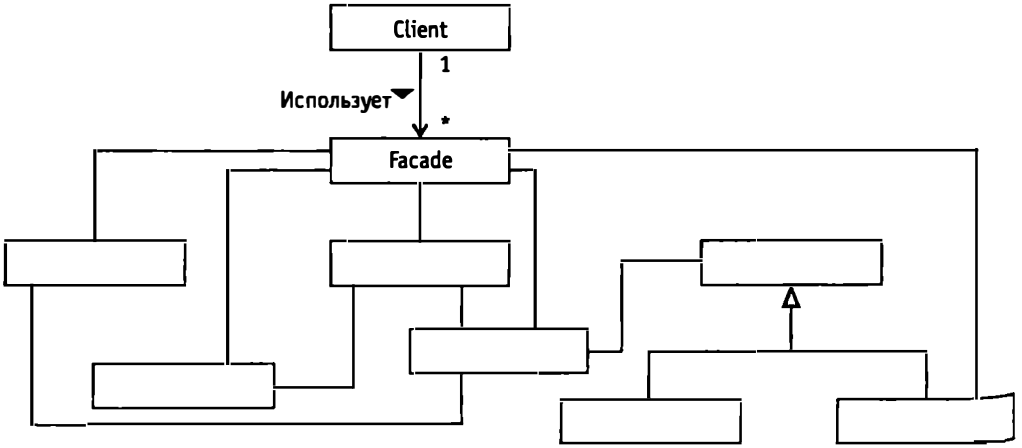


Рис. 7.12. Шаблон Facade

Нет никакой необходимости в том, чтобы класс Facade являлся непроницаемым барьером, отделяющим клиентские классы от классов, реализующих абстракцию. Вполне достаточно (а иногда и лучше), если класс Facade просто будет представлять собой задаваемый по умолчанию способ доступа к функциональности классов, реализующих абстракцию. Если некоторые клиенты должны иметь прямой доступ к классу, реализующему абстракцию, то класс Facade может иметь метод, который возвращает ссылку на соответствующий объект реализации.

Назначение класса Facade состоит в том, чтобы разрешить использование простых клиентов, а не требовать их в обязательном порядке.

РЕАЛИЗАЦИЯ

Класс Facade должен предоставлять клиентским объектам способ получения прямой ссылки на экземпляр реализующего абстракцию класса, в котором клиентские объекты могут нуждаться. Однако могут существовать такие реализующие абстракцию классы, о которых клиентские классы не должны ничего знать. Класс Facade обязан скрывать такие классы от клиентских классов. Это можно осуществить следующим образом: такие классы делаются закрытыми внутренними классами класса Facade.

Иногда нужно изменить классы реализации, используемые объектом-фасадом для согласования изменений реализуемой абстракции. Например (возвращаясь к ситуации с электронной почтой, описанной в разделе «Контекст»), могут потребоваться разные наборы классов для создания сообщений, соответствующих стандартам MIME, MAPI или Notes. Разные наборы классов реализаций обычно требуют разных классов Facade. Можно скрыть от клиентских классов факт использования различных классов Facade, применяя шаблон Interface (см. гл. 4): определить интерфейс, который должны реализовывать все классы Facade, участвующие в создании электронной почты. Затем заставить клиентские классы обращаться к классу Facade не прямым образом, а через интерфейс.

СЛЕДСТВИЯ

- ☺ Если поместить класс-фасад между классами, реализующими абстракцию, и их клиентами, то упростятся клиентские классы, так как зависимости клиентских классов передадутся классу-фасаду. Клиентам объектов-фасадов не нужно знать о каких-либо классах, скрывающихся за классом-фасадом.
- ☺ Шаблон Facade уменьшает или устраняет связь между клиентским классом и реализующими абстракцию классами, поэтому существует возможность изменять классы, реализующие абстракцию, не затрагивая клиентские классы.

ПРИМЕНЕНИЕ В JAVA API

В java.net класс URL может служить примером шаблона Facade. Он обеспечивает доступ к содержимому ресурсов, на которые указывают URL. Класс может быть клиентом класса URL и использовать его для получения содержимого URL, не будучи осведомленным о множестве классов, находящихся за фасадом, который предоставляется классом URL. С другой стороны, чтобы послать данные на URL, клиент объекта URL может вызвать его метод openConnection, который возвращает объект URLConnection, используемый объектом URL.

ПРИМЕР КОДА

Рассмотрим код для класса MessageCreator, представленного на рис. 7.11. Экземпляры класса MessageCreator используются для создания и отправления сообщений электронной почты. Он представлен здесь как типичный пример класса Facade.

```
public class MessageCreator {
   // Константы, обозначающие тип создаваемых сообщений.
   public final static int MIME = 1;
   public final static int MAPI = 2;
...
   private Hashtable headerFields = new Hashtable();
   private RichText messageBody;
   private Vector attachments = new Vector();
   private boolean signMessage;

   public MessageCreator(String to,
                         String from,
                         String subject) {
      this(to, from , subject, inferMessageType(to));
   } // Constructor(String, String, String)

   public MessageCreator(String to, String from,
                         String subject, int type) {
      headerFields.put("to", to);
      headerFields.put("from", from);
      headerFields.put("subject", subject);
      ...
   } // Constructor(String, String, String, int)

   /**
    * Задаем содержимое тела сообщения.
    */
```

```
public void setMessageBody(String messageBody) {
   setMessageBody(new RichTextString(messageBody));
} // setMessageBody(String)

/**
 * Задаем содержимое тела сообщения.
 */
public void setMessageBody(RichText messageBody) {
   this.messageBody = messageBody;
} // setMessageBody(RichText)

/**
 * Добавляем к сообщению присоединяемую информацию.
 */
public void addAttachment(Object attachment) {
   attachments.addElement(attachment);
} // addAttachment(Object)

/**
 * Указываем, будет ли сообщение подписано.
 * Значение по умолчанию — false.
 */
public void setSignMessage(boolean signFlag) {
   signMessage = signFlag;
} // setSignMessage(boolean)

/**
 * Задаем значение для поля заголовка.
 */
public void setHeaderField(String name, String value) {
   headerFields.put(name.toLowerCase(), value);
} // setHeaderField(String, String)

/**
 * Отправляем сообщение.
 */
public void send() {
   ...
} // send()
```

```
    /**
     * Выясняем тип сообщения, исходя из адреса назначения
     * электронной почты.
     */
    private static int inferMessageType(String address) {
       int type = 0;
    ...
       return type;
    } // inferMessageType(String)

    /**
     * Создаем объект Security, чтобы подписать это сообщение.
     */
    private Security createSecurity() {
       Security s = null;
       ...
       return s;
    } // createSecurity()

    /**
     * Создаем объект MessageSender,
     * соответствующий типу отправляемого сообщения.
     */
    private void createMessageSender(Message msg) {
       ...
    } // createMessageSender(Message))
...
} // class MessageCreator
```

Шаблон Facade не предъявляет никаких требований к классам, которые используются классом Facade. Поскольку они не добавляют ничего нового информацию о шаблоне Facade, их исходный текст здесь не приводится.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ FACADE

nterface. Можно использовать шаблон Interface вместе с шаблоном Facade. Это позволит применять различные классы фасадов, не требуя от клиентских классов осведомленности о разных классах.

.aw of Demeter. Концептуальная модель, применяющая шаблон Law of Demeter описанный в книге [Grand99]), часто дает начало проекту, который использует шаблон Facade.

Adapter. Шаблон Adapter позволяет клиентским классам считать единственный объект, не реализующий некоторый интерфейс, объектом, действительно реализующим этот интерфейс. Шаблон Facade позволяет клиентским классам рассматривать группу объектов как единственный объект, реализующий определенный интерфейс.

Pure Fabrication. Проект класса фасада представляет собой случай применения шаблона Pure Fabrication, описанного в книге [Grand99].

Flyweight (Приспособленец)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Если экземпляры класса, содержащие одинаковую информацию, могут использоваться как взаимозаменяемые, то шаблон Flyweight позволяет программе избежать издержек, связанных с созданием множества экземпляров, содержащих одинаковую информацию, применяя механизм совместного использования одного экземпляра.

КОНТЕКСТ

Предположим, создается текстовый процессор. На рис. 7.13 представлена диаграмма классов, описывающая некоторые основные классы, которые можно использовать для представления документа:

- `DocumentElement` —исходный суперкласс для всех классов, используемых для представления документа; все подклассы класса `DocumentElement` наследуют методы для записи и считывания своих шрифтов;
- экземпляр класса `DocChar` представляет каждый символ документа;
- `DocumentContainer` — суперкласс для контейнерных классов `Document`, `Page`, `Paragraph` и `LineOfText`.

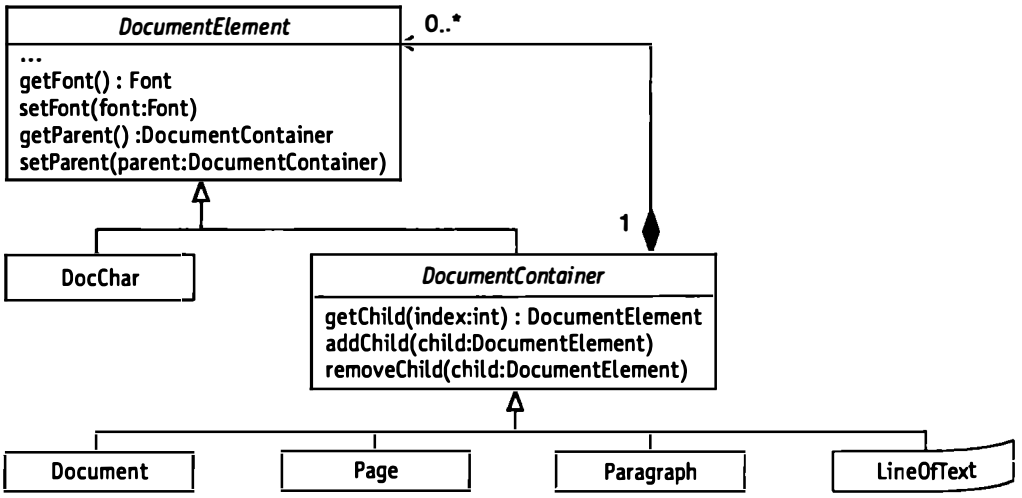


Рис. 7.13. Классы представления документа

Можно задать шрифт для каждого символа, вызывая метод `setFont` объекта `DocChar`, представляющего этот символ. Если шрифт символа не задан, то он использует шрифт контейнера. Если шрифт контейнера не был задан, то контейнер использует шрифт своего контейнера и т.д.

Один документ, состоящий из нескольких страниц, может содержать десятки объектов `Paragraph`, которые содержат несколько сотен объектов `LineOfText` и тысячи или десятки тысяч объектов `DocChar`. Очевидно, что результатом использования такого проекта будет программа, требующая значительного объема памяти для хранения символов.

Существует возможность избежать больших затрат памяти на эти многочисленные объекты `DocChar`, имея только один экземпляр каждого отдельного объекта `DocChar`. Изображенные на рис. 7.13 классы применяют объект `DocChar` для представления каждого символа документа. Чтобы представить фразу «She saw her father», объект `LineOfText` использует объекты `DocChar` (рис. 7.14).

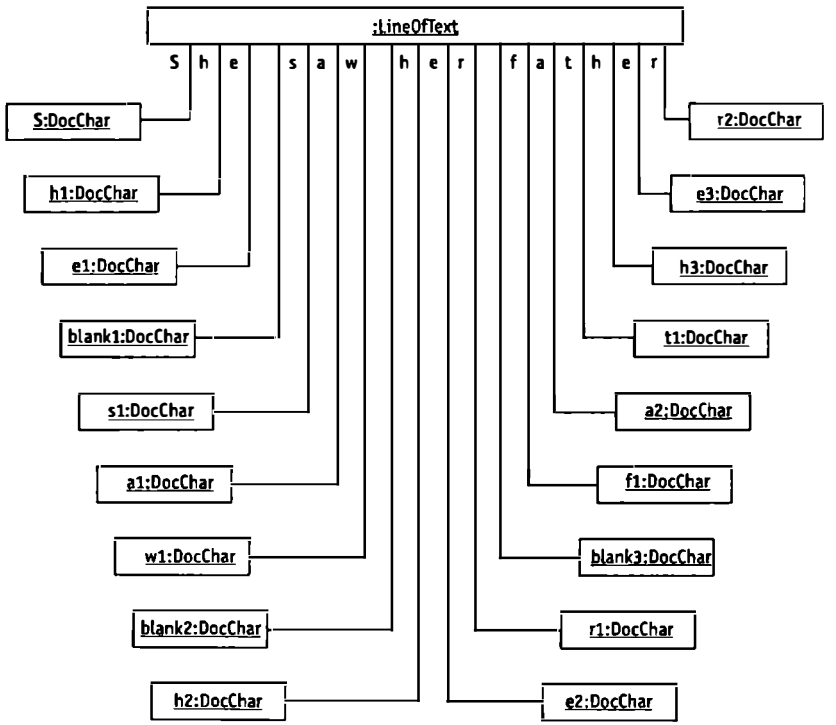


Рис. 7.14. Объекты символов, не являющиеся совместно используемыми

Отметим, что в строке несколько раз встречаются символы «h», «e», « » (пробел), «a» и «r». В документе все символы обычно встречаются много раз. Можно реорганизовать объекты таким образом, чтобы один объект `DocChar` представлял все случаи появления одного и того же символа (рис. 7.15).

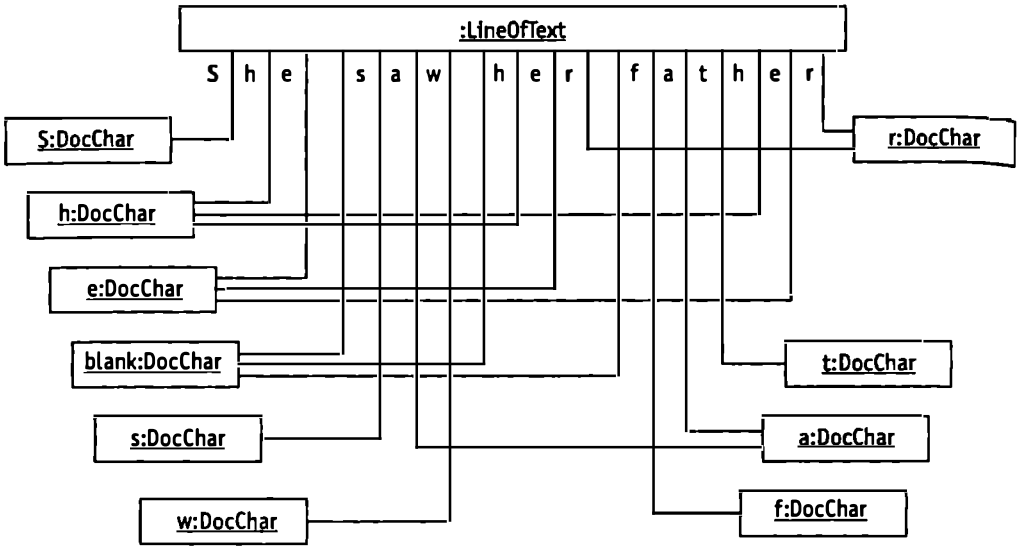


Рис. 7.15. Совместно используемые объекты символов

Чтобы быть совместно используемыми, объекты `DocChar` не должны обладать какими-либо внутренними атрибутами, которые не являются общими для любого места, на которое ссылается объект. Внутренний атрибут — это атрибут, значение которого хранится вместе с объектом. Он отличается от внешнего атрибута, значение которого хранится вне своего объекта.

Организация классов, представленная на рис. 7.13, описывает класс `DocChar`, экземпляры которого могут иметь внутренний атрибут шрифта. Те объекты `DocChar`, которые не имеют храняшегося внутри них шрифта, используют шрифт своего абзаца.

Чтобы совместно использовать объекты `DocChar`, классы должны быть реорганизованы таким образом, чтобы объекты `DocChar`, имеющие свои собственные шрифты, хранили бы их в виде внешних атрибутов. На рис. 7.16 представлен также класс `CharacterContext`, экземпляры которого сохраняют внешние атрибуты для целого ряда символов.

В этой структуре класс `DocCharFactory` отвечает за предоставление объекта `DocChar`, определяющего данный символ. Если задавать для представления один и тот же символ, метод `getDocChar` объекта `DocCharFactory` всегда будет возвращать один и тот же объект `DocChar`. Кроме того, класс `DocumentContainer`, а не класс `DocumentElement`, определяет методы по работе со шрифтом. Все конкретные классы, за исключением `DocChar`, являются подклассами класса `DocumentContainer`. Это означает, что класс `DocChar` не имеет внутреннего атрибута шрифта. Если пользователь захочет связать шрифт с каким-либо символом или набором символов, то программа создаст объект `CharacterContext` (рис. 7.17).

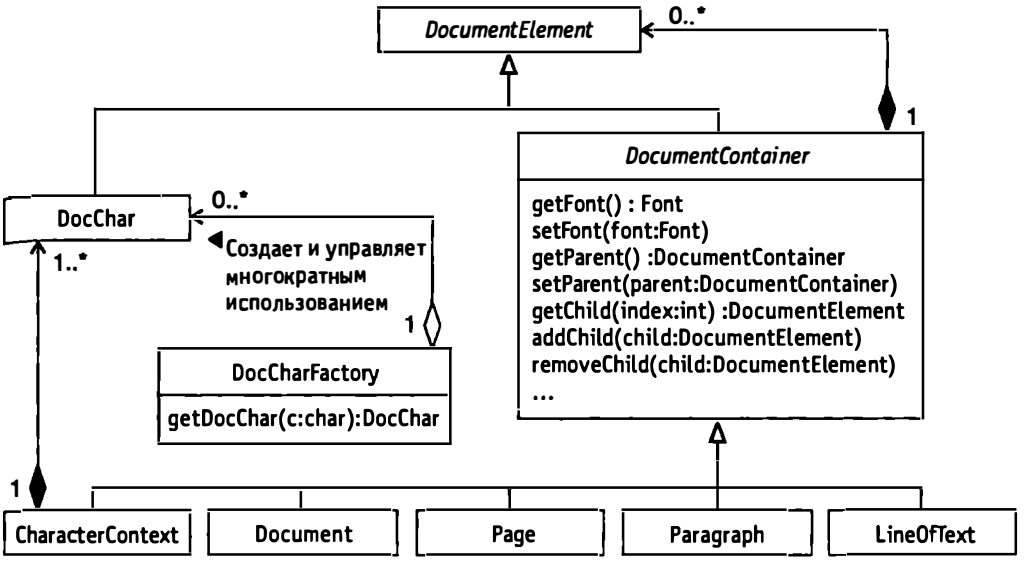


Рис. 7.16. Совместно используемые классы представления документа

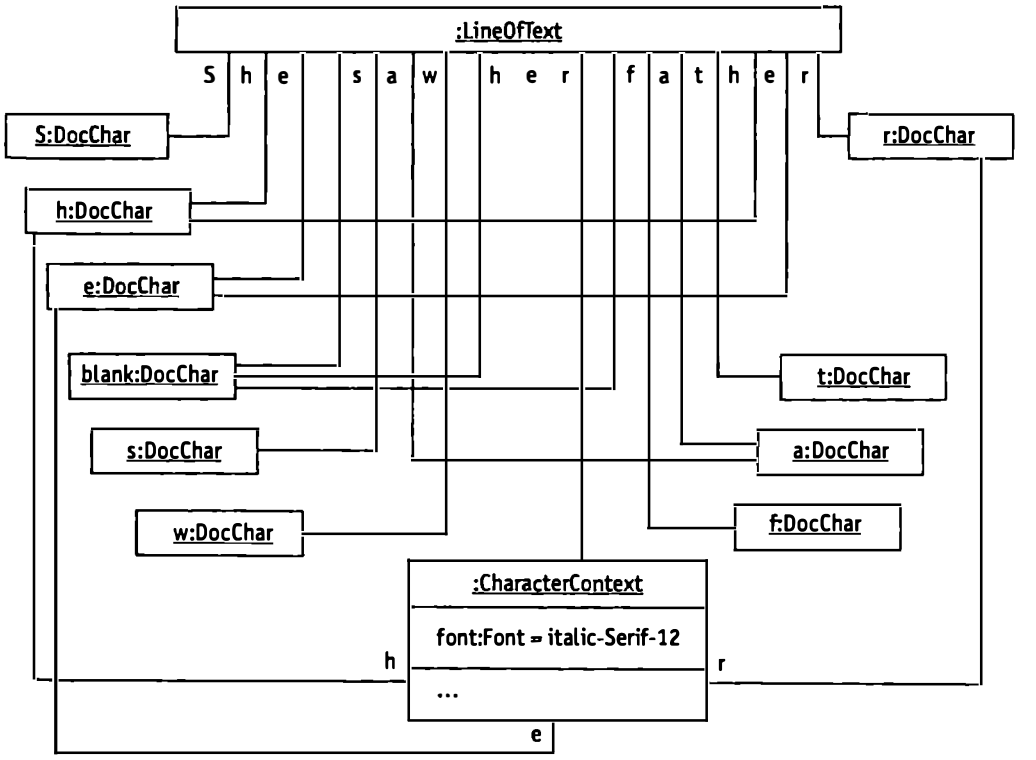


Рис. 7.17. Шрифт в объекте CharacterContext

МОТИВЫ

☺ Есть приложение, которое использует большое количество одинаковых объектов.

☺ Необходимо уменьшить значительные затраты памяти на поддержку большого количества похожих объектов.

☺ Программа не полагается на уникальность объектов любых объектов, которые она, по желанию разработчика, должна совместно использовать. Если программа применяет различные объекты в разных контекстах, то существует возможность различать контексты, используя уникальность объектов. Если различные контексты совместно используют одни и те же объекты, то эти контексты уже нельзя отличать при помощи уникальности объектов.

☺ Представление похожих вещей похожими объектами требует большего объема памяти, чем представление похожих вещей при помощи одного и того же объекта. Чем больше вещей могут быть представлены одним и тем же объектом, тем значительнее экономия памяти.

РЕШЕНИЕ

На рис. 7.18 представлена основная организация классов для шаблона Flyweight.

Опишем роли, которые исполняют эти классы.

AbstractFlyweight. Класс `AbstractFlyweight` представляет собой суперкласс для всех других классов-приспособленцев. Он определяет общие для классов-приспособленцев методы. Эти методы нуждаются в доступе к информации о внешнем состоянии и обычно получают его с помощью параметров.

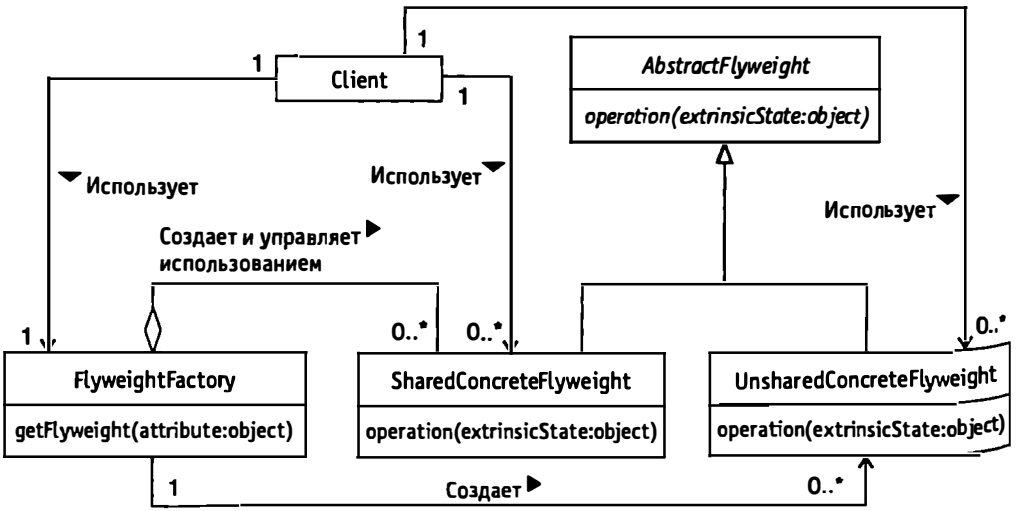


Рис. 7.18. Шаблон Flyweight

SharedConcreteFlyweight. Экземпляры классов в этой роли являются объектами общего использования. Если они содержат какое-либо внутреннее состояние, то оно должно быть общим для всех представляемых ими сущностей. Например, совместно используемые объекты `DocChar`, описанные в примере раздела «Контекст», представляют символ как свое внутреннее состояние.

UnsharedConcreteFlyweight. Экземпляры классов, участвующих в роли `UnsharedConcreteFlyweight`, не являются совместно используемыми. Шаблон Flyweight не требует от объектов быть совместно используемыми. Он просто разрешает совместное использование объектов. Если существуют объекты, которые не используются совместно и являются экземплярами класса `AbstractFlyweight`, то они не будут экземплярами `SharedConcreteFlyweight` подклассов класса `AbstractFlyweight`.

FlyweightFactory. Экземпляры классов `FlyweightFactory` предоставляют экземпляры класса `AbstractFlyweight` клиентским объектам. Когда клиентский объект запрашивает объект `FlyweightFactory` с целью предоставления экземпляра класса `UnsharedConcreteFlyweight`, тот просто создает этот экземпляр. Однако, когда клиентский объект обращается к объекту `FlyweightFactory` за предоставлением экземпляра класса `SharedConcreteFlyweight`, тот сначала проверяет, не был ли похожий объект создан ранее. Если такой объект уже был создан, то он предоставляет клиентскому объекту ранее созданный объект. В противном случае он создает новый объект и предоставляет его клиенту.

Client. Экземпляры классов `Client` — это объекты, использующие объекты-приспособленцы.

Если в роли `SharedConcreteFlyweight` выступает только один класс, то могут не понадобиться классы, исполняющие роль `AbstractFlyweight` или `UnsharedConcreteFlyweight`.

РЕАЛИЗАЦИЯ

Между количеством атрибутов, которые создают внешними, и количеством объектов-приспособленцев, необходимых на стадии выполнения, существует некоторый компромисс. Чем больше атрибутов создают внешними, тем меньше потребуется объектов-приспособленцев. Чем больше атрибутов создают внутренними, тем меньше времени потратят объекты для доступа к своим атрибутам.

Рассмотрим пример представления документа. Если пользователь, например, выделяет несколько символов курсивом, то программа создаст отдельный объект `CharacterContext`, который будет содержать внешний атрибут шрифта для тех объектов `DocChar`, которые представляют эти символы. Альтернативный вариант позволяет атрибуту шрифта быть внутренним атрибутом объектов `DocChar`. Если атрибут шрифта является внутренним, то объекты `DocChar` будут затрачивать на доступ к своему атрибуту шрифта меньше времени. Если атрибуту шрифта разрешено быть внутренним, то это означает также, что программе потребуется отдельный объект `DocChar` для представления каждой комбинации «символ — шрифт».

СЛЕДСТВИЯ

☺ Применяя объекты-приспособленцы совместного использования, можно значительно уменьшить количество объектов, находящихся в памяти.

☹ Шаблон Flyweight усложняет программу. Основными источниками дополнительной сложности являются объекты-приспособленцы со своими внешними состояниями, а также управление многократным использованием объектов-приспособленцев.

☹ Шаблон Flyweight может привести к увеличению времени работы программы, так как доступ объекта к внешнему состоянию требует больших усилий, чем доступ к внутреннему состоянию.

☹ Обычно можно различать сущности по объектам, которые их представляют. Шаблон Flyweight делает это невозможным, так как с его помощью множество сущностей представляется в конечном счете одним и тем же объектом.

☹ Совместно используемые объекты-приспособленцы не могут содержать ссылок на родителей.

☹ Из-за сложности, связанной с использованием шаблона Flyweight, и ограничений, накладываемых им на организацию классов, следует рассматривать шаблон Flyweight как оптимизацию, применяемую после разработки остальной части проекта.

ПРИМЕНЕНИЕ В JAVA API

Java использует шаблон Flyweight для управления объектами `String`, которые применяются для представления строковых литералов. Если в программе имеется более одного строкового литерала и эти литералы содержат одну и ту же последовательность символов, то Java использует один и тот же объект `String` для представления всех этих литералов.

Метод `intern` класса `String` отвечает за управление объектами `String`, которые используются для представления строковых литералов.

ПРИМЕР КОДА

Приведем код, который реализует диаграмму классов, изображенную на рис. 7.16. Исходный текст некоторых классов не представляет интереса с точки зрения шаблона Flyweight, поэтому коды таких классов опущены, например, класса `DocumentElement`. Класс `DocumentContainer` определяет некоторые методы, которые наследуют все контейнерные классы, применяемые для представления документа:

```
abstract class DocumentContainer extends DocumentElement {
   // Коллекция потомков этого объекта.
   private Vector children = new Vector();
```

```
// Это шрифт, связанный с данным объектом.
// Если переменная шрифта — null, то шрифт этого объекта
// будет наследоваться от контейнера этого объекта
// через иерархию контейнеров.
private Font font;

DocumentContainer parent; // Контейнер этого объекта.

/**
 * Возвращает потомка этого объекта, под номером index.
 */
public DocumentElement getChild(int index) {
   return (DocumentElement)children.elementAt(index);
} // getChild(int)

/**
 * Делаем данный DocumentElement потомком этого объекта.
 */
public synchronized void addChild(DocumentElement child) {
   synchronized (child) {
      children.addElement(child);
      if (child instanceof DocumentContainer)
         ((DocumentContainer)child).parent = this;
   } // synchronized
} // addChild(DocumentElement)

/**
 * Делаем так, чтобы данный DocumentElement НЕ БЫЛ потомком
 * этого объекта.
 */
public
synchronized void removeChild(DocumentElement child) {
   synchronized (child) {
      if (child instanceof DocumentContainer
            && this == ((DocumentContainer)child).parent)
         ((DocumentContainer)child).parent = null;
      children.removeElement(child);
   } // synchronized
} // removeChild(DocumentElement)

/**
 * Возвращает родителя этого объекта или null (если у него нет
 * родителя).
```

```
     */
    public DocumentContainer getParent() {
       return parent;
    } // getParent()

    /**
     * Возвращает шрифт, связанный с этим объектом.
     * Если нет шрифта, связанного с этим объектом, то возвращает
     * шрифт его родителя. Если нет шрифта, связанного
     * с родителем этого объекта, то возвращает null.
     */
    public Font getFont() {
       if (font != null)
          return font;
       else if (parent != null)
          return parent.getFont();
       else
          return null;
    } // getFont()

    /**
     * Связываем  шрифт с этим объектом.
     */
    public void setFont(Font font) {
       this.font = font;
    } // setFont(Font)
...
} // class DocumentContainer
```

Методы, описанные в классе `DocumentContainer`, управляют состоянием всех контейнерных классов документа, включая класс `CharacterContext`. Используя эти унаследованные методы, класс `CharacterContext` способен управлять внешним состоянием объектов `DocChar`, даже если он не объявлял с этой целью какие-либо собственные методы. Приведем код класса `DocChar`, который представляет символы документа:

```
class DocChar extends DocumentElement {
   private char character;

   DocChar (char c) {
      character = c;
   } // Constructor(char)
```

```
    /**
     * Возвращает символ, который представляет этот объект.
     */
    public char getChar() {
      return character;
    } // getChar()

    /**
     * Этот метод возвращает уникальное значение, которое
     * определяет место внутреннего хранения в хэш-таблице.
     */
    public int hashCode() {
      return getChar();
    } // hashCode()

    /**
     * Переопределяем равенство таким образом, что два объекта
     * DocChar считаются равными, если они представляют
     * один и тот же символ.
     */
    public boolean equals(Object o) {
      // Не прямым образом обращаемся к символу, а вызываем
      // метод getChar  с тем, чтобы этот метод соответствовал
      // какому-либо имеющемуся у подкласса альтернативному
      // методу предоставления описываемого им символа.
      return (o instanceof DocChar
              && ((DocChar)o).getChar() == getChar());
    } // equals(Object)
  } // class DocChar
```

И наконец, приведем код для класса `DocCharFactory`, который отвечает за совместное использование объектов `DocChar`:

```
  class DocCharFactory {
    private MutableDocChar myChar = new MutableDocChar();

    /**
     * Коллекция ранее созданных объектов DocChar.
     */
    private HashMap docCharPool = new HashMap();

    /**
     * Возвращает объект DocChar, представляющий данный символ.
```

```
     */
    synchronized DocChar getDocChar(char c) {
      myChar.setChar(c);
      DocChar thisChar = (DocChar)docCharPool.get(myChar);
      if (thisChar == null) {
          thisChar = new DocChar(c);
          docCharPool.put(thisChar, thisChar);
      } // if
      return thisChar;
    } // getDocChar(char)
```

Чтобы разрешить поиск объектов DocChar в HashMap или аналогичной коллекции, следует предъявить коллекции объект DocChar, описывающий тот же символ, что и объект DocChar, который нужно найти в коллекции. Создание объекта DocChar при выполнении каждой операции поиска означало бы, что все еще создается объект DocChar для каждого символа документа. Хотя эти объекты DocChar были бы удалены программой сборки мусора, поскольку после их участия в операции поиска на них нет больше ссылок, но в первую очередь все же лучше вообще отказаться от их создания.

Альтернативой созданию объекта DocChar при выполнении каждой операции поиска должно стать многократное использование одного и того же объекта DocChar при условии изменения представляемого им символа для каждого нового поиска. Проблема, возникающая при изменении описываемого объектом DocChar символа, состоит в том, что объекты DocChar являются неизменными, и не существует способа изменить символ, представляемый объектом DocChar.

Класс DocCharFactory выходит из этого затруднения, используя такой закрытый подкласс класса DocChar, который предоставляет способ изменения описываемого им символа:

```
private static class MutableDocChar extends DocChar {
    private char character;

    MutableDocChar() {
    // Не имеет значения, что передаем конструктору.
    super('\u0000');
    } // Constructor(char)

    /**
    * Возвращает символ, описываемый этим объектом.
    */
    public char getChar() {
       return character;
```

```
    } // getChar()

    /**
    * Задаем символ,  описываемый этим объектом.
    */
    public void setChar(char c) {
       character = c;
    } // setChar(char)
  } // class MutableDocChar
} // class DocCharFactory
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ FLYWEIGHT

Composite. Шаблон Flyweight часто используется вместе с шаблоном Composite для представления листьев иерархической структуры при помоши совместно используемых объектов.

Factory Method. Шаблон Flyweight использует шаблон Factory Method для создания новых объектов-приспособленцев.

Cache Management. Реализация класса `FlyweightFactory` может использовать кэш.

Immutable. Совместно используемые объекты-приспособленцы часто бывают неизменными.

Dynamic Linkage (Динамическая компоновка)

СИНОПСИС

Позволяет программе в ответ на запрос загружать и использовать произвольные классы, реализующие известный интерфейс.

КОНТЕКСТ

Предположим, создается программа для нового вида интеллектуального пищевого процессора, в который можно загружать сырые ингредиенты, и он, нарезая ломтиками, кубиками, смешивая, отваривая, выпекая, зажаривая и помешивая, способен производить готовую к употреблению пищу. На механическом уровне новый пищевой процессор — очень сложное оборудования. Однако для пищевого процессора очень важен выбор программ для приготовления различной пищи. Программа, при помощи которой можно смешивать муку, воду, дрожжи и другие ингредиенты для получения различных видов хлеба, значительно отличается от программы, при помощи которой можно поджаривать, помешивая, креветки в строгом соответствии с нужным рецептом. Пищевой процессор должен запускать множество различных программ, которые позволяют ему готовить разнообразную пищу. При этом невозможно встроить в пищевой процессор все необходимые программы, поэтому пищевой процессор должен загружать программы с CD-ROM или аналогичных носителей.

Чтобы такие динамически загружаемые программы и операционная среда пищевого процессора могли работать вместе, им нужен способ, позволяющий вызывать методы друг друга. На рис. 7.19 представлена диаграмма классов, реализующая такую возможность.

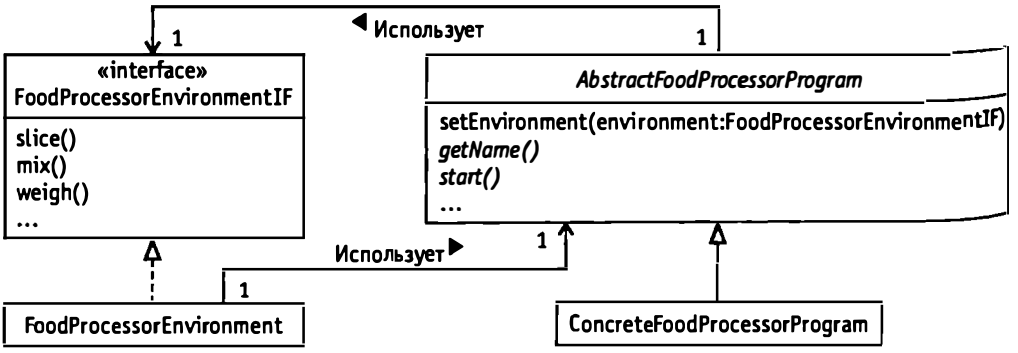


Рис. 7.19. Классы для программы пищевого процессора

Такая организация позволяет объекту, находящемуся в среде пищевого процессора, вызывать методы объекта верхнего уровня, находящегося в программе пищевого процессора, обращаясь к методам его суперкласса. Кроме того, объект верхнего уровня может вызывать методы объекта среды процессора через реализуемый им интерфейс `FoodProcessorEnvironmentIF`. На рис. 7.20 изображена диаграмма взаимодействия, демонстрирующая совместную работу этих классов. Здесь представлены начальные шаги, предпринимаемые в тот момент, когда операционная среда пищевого процессора получает запрос на запуск программы.

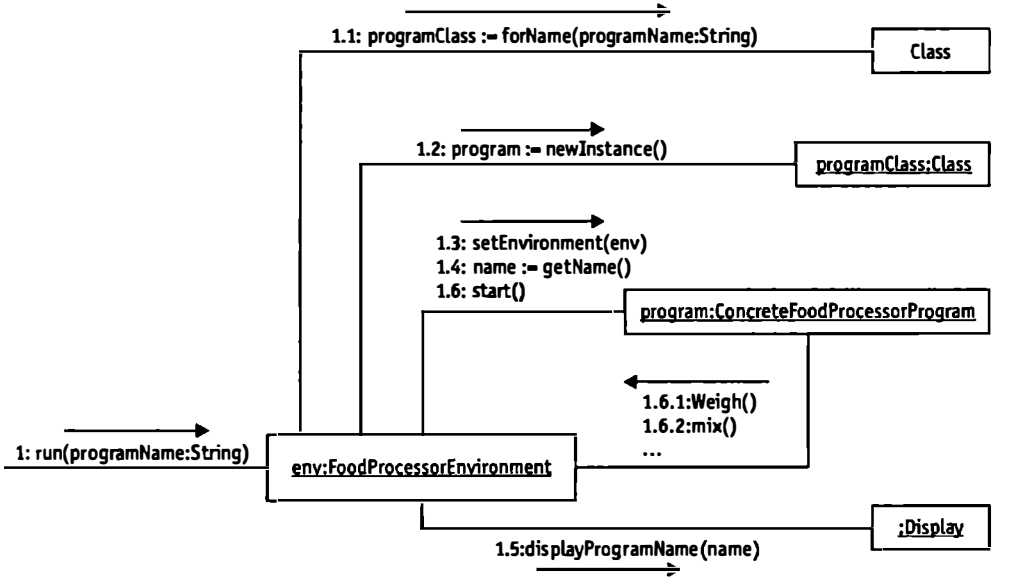


Рис. 7.20. Взаимодействие классов пищевого процессора

1. Операционная среда пищевого процеса получает запрос на запуск программы с заданным именем.

   1.1. Среда вызывает метод `forName` класса `Class`, передавая ему имя запускаемой программы. Метод `forName` находит объект `Class` (с таким же, как и у программы, именем). В случае необходимости он загружает класс с CD-ROM. Метод `forName` завершается, возвращая объект `Class`, в котором инкапсулирован класс верхнего уровня этой программы.

   1.2. Среда создает для программы экземпляр класса верхнего уровня. На диаграмме этот экземпляр указан под именем `program`.

   1.3. Среда передает ссылку на саму себя методу `setEnvironment` объекта `program`. Если программе передана такая ссылка, программа может обращаться к методам операционной среды.

1.4. Из программы операционная среда получает имя программы.

1.5. Операционная среда отображает имя программы на экране.

1.6. Операционная среда запускает программу.

1.6.1. Программа взвешивает описанные в ней ингредиенты.

1.6.2. Программа смешивает описанные в ней ингредиенты.

Программа продолжает работать, выполняя последующие шаги, которые уже не имеют отношения к данной схеме.

МОТИВЫ

☺ Программа должна иметь возможность загружать и использовать произвольные классы, о которых она ничего не знает заранее.

☺ Экземпляры загруженного класса должны иметь возможность обратного вызова программы, загрузившей его.

☹ Добавление в программу классов, которые не были предусмотрены при ее создании, связано с риском потери безопасности. Кроме того, существует риск несоответствия класса и программы ввиду несовпадения используемых версий.

РЕШЕНИЕ

На рис. 7.21 показана диаграмма классов, на которой представлены роли интерфейсов и классов, принимающих участие в шаблоне Dynamic Linkage.

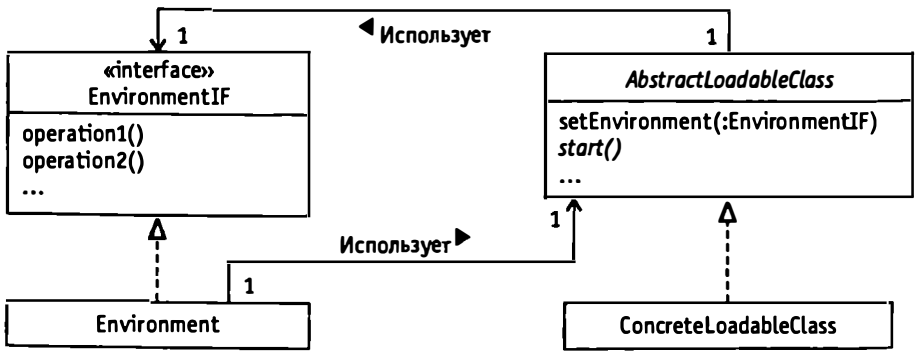


Рис. 7.21. Шаблон Dynamic Linkage

Опишем эти роли.

EnvironmentIF. Интерфейс, выступающий в этой роли, объявляет методы среды, которые могут быть вызваны загруженным классом.

Environment. Класс в этой роли представляет собой часть среды, которая загружает класс `ConcreteLoadableClass`. Он реализует интерфейс `EnvironmentIF`. Ссылка на экземпляр этого класса передается экземплярам класса `ConcreteLoadableClass`, поэтому они могут вызывать методы объекта `Environment`, объявляемые интерфейсом `EnvironmentIF`.

AbstractLoadableClass. Любой класс верхнего уровня, объявленный в программе пищевого процессора, должен быть подклассом некоторого класса, выступающего в роли `AbstractLoadableClass`. Считается, что класс в этой роли объявляет другие, как правило, абстрактные методы в дополнение к двум следующим методам.

1. Метод, с именем, например, `setEnvironment`, который позволяет экземплярам подклассов класса `AbstractLoadableClass` получить ссылку на экземпляр класса, реализующего интерфейс `EnvironmentIF`. Назначение этого метода состоит в том, чтобы позволить объектам `AbstractLoadableClass` вызывать методы объекта `Environment`.
2. Метод, обычно с именем `start`, вызывается средой, чтобы сообщить экземпляру загруженного класса о начале его работы.

ConcreteLoadableClass. Классы в этой роли являются динамически загружаемыми подклассами класса `AbstractLoadableClass`.

РЕАЛИЗАЦИЯ

Представленный шаблон Dynamic Linkage считает, что среда должна знать о классе `AbstractLoadableClass` и что загруженный класс должен знать об интерфейсе `EnvironmentIF`. В тех случаях, где не нужна такая большая структура, возможны другие механизмы взаимодействия. Например, технология JavaBeans использует комбинацию из механизма отражения классов и соглашений об именах с той целью, чтобы позволить другим классам вынести решение о способе взаимодействия с бином (bean).

Другое требование шаблона Dynamic Linkage состоит в том, что класс `Environment` должен каким-то образом узнать имя класса, который он хочет загрузить. Механизм выяснения имени зависит от приложения. В некоторых случаях имена могут быть заданы с помощью аппаратных средств. В примере с пищевым процессором должен существовать соответствующий механизм, определяющий расположение катологов программ на CD-ROM или другом дистрибутивном носителе. Пищевой процессор показывает каталог программ в виде меню, позволяя пользователю сделать выбор.

Несовместимые классы

Некоторые случаи реализации шаблона Dynamic Linkage должны принимать во внимание то, что различные динамически загруженные классы могут использовать несовместимые версии одного и того же класса. Предположим, что программы приготовления лазаньи и пельменей поступают на разных носителях

и обе используют утилитный класс `Fu`. Однако два класса с именем `Fu` несовместимы друг с другом. Допустим, что пищевой процессор сначала запускает программу приготовления лазаньи, а затем пытается запустить программу изготовления пельменей. Если при загрузке программы по изготовлению пельменей ему предоставляется класс `Fu` для программы по изготовлению лазаньи, программа по изготовлению пельменей работать не будет.

Стратегия решения этой проблемы заключается в том, чтобы гарантировать, что все вспомогательные классы, неявно динамически загруженные вместе с явно динамически загруженным классом, не использовались бы каким-либо другим явно динамически загруженным классом. Можно реализовать эту стратегию, используя для каждого динамически загруженного класса свой объект `ClassLoader`. Например, некоторые браузеры используют разные загрузчики классов для загрузки разных апплетов. При этом классы, загруженные как часть одного апплета, уже не могут использоваться как часть другого.

Риск, связанный с нарушением безопасности

Работа программы, которая динамически загружает классы и вызывает их методы, связана с некоторым риском нарушения безопасности. Нужно, чтобы загруженные классы вели себя надлежащим образом. Они, как правило, так и делают, однако существует вероятность и другого хода событий. Опишем некоторые возможные риски.

- Класс может сделать нечто такое, что не позволит использовать операционную среду для других целей. Один из его методов, который должен был закончить выполнение спустя какое-то время, может не сделать это никогда. Если среда является многопоточной, этот класс может использовать время центрального процессора, память и иные ресурсы, необходимые для других целей.
- Класс может делать что-то, что нарушает целостность его среды. Такие нарушения могут принимать самые разные формы.

Данная книга не предусматривает подробное описание способов решения подобных проблем. Здесь даются только некоторые общие замечания, которые позволят оказаться на верном пути.

Первый, требующий решения, вопрос заключается в том, чтобы оценить величину риска в конкретном случае. Если вероятность неправильного хода событий достаточно мала или последствия достаточно незначительны, то не стоит предпринимать каких-либо мер, чтобы уменьшить вероятность или последствия возникновения проблем с безопасностью. С другой стороны, если вероятность плохого хода событий или последствия достаточно велики, то следует принять все возможные меры.

Следующий вопрос, какова вероятность того, что какой-то конкретный класс будет причиной появления проблемы нарушения безопасности. В таком случае часто используют термин *доверие*. Основная проблема, связанная с доверием,

состоит в том, можно или нельзя разрешить использовать класс в некоторой среде. В Java за загрузку классов отвечает подкласс абстрактного класса `java.lang.ClassLoader`. Документация к этому классу содержит подробное описание механизма загрузки классов.

Общая стратегия, применяющаяся с целью определения уровня доверия к классу, основана на уровне доверия к компании, которая создала этот класс. Можно определить, откуда пришел класс, если он снабжен сертификатом, содержащим цифровую подпись компании, которая создала этот класс. Обычно классы поставляются в jar-файлах, имеющих цифровые подписи их создателей.

Java обладает тщательно разработанным механизмом контроля доступа к методам, основанном на доверии и полномочиях. Он описан в документе «Java 2 Platform Security Architecture», который входит в комплект документации, поставляемый компанией Sun вместе с Java 2 SDK.

Надлежащее использование механизмов, предоставляемых архитектурой системы безопасности, не позволит классам, полученным из источников, которым не доверяют, делать то, что они не должны делать. В качестве дополнительной меры можно использовать защитный прокси-класс, описанный в книге [Grand2001]. Таким образом даже классы, которым доверяют, могут получить доступ только к определенным вещам.

СЛЕДСТВИЯ

☺ Подклассы класса `AbstractLoadableClass` могут загружаться динамически.

☺ Операционная среда и загруженные классы не нуждаются в какой-либо определенной предварительной информации друг о друге.

• Динамическая компоновка увеличивает общее время, которое нужно программе для загрузки всех используемых ею классов. Однако здесь может иметь место эффект распределения расходов на загрузку с течением времени. С этой целью уместно использование шаблона Virtual Proxy.

☹ Использование шаблона Dynamic Linkage связано с риском нарушения системы безопасности.

ПРИМЕНЕНИЕ В JAVA API

Web-браузеры используют шаблон Dynamic Linkage для запуска апплетов. На рис. 7.22 представлена диаграмма классов, описывающая отношения между апплетом и браузером.

Операционная среда браузера имеет доступ к загруженному ею подклассу класса `Applet` как к экземпляру класса `Applet`. Загруженные подклассы апплета получают доступ к среде браузера через интерфейс `AppletStub`.

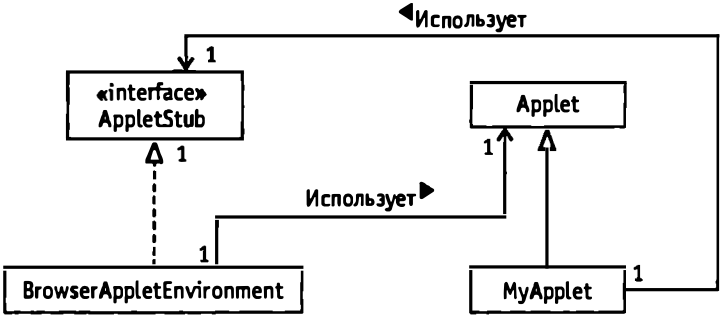


Рис. 7.22. Апплеты и браузеры

ПРИМЕР КОДА

Приведем код, который реализует проект пищевого процессора, описанный в разделе «Контекст». Интерфейс для операционной среды пищевого процессора:

```
public interface FoodProcessorEnvironmentIF {
   /**
    * Нарезает пищу ломтиками заданной ширины.
    */
   public void slice(int width) ;

   /**
    * Перемешивает пищу с заданной скоростью.
    */
   public void mix(int speed) ;

   /**
    * Взвешивает пищу.
    * Возвращает вес в унциях.
    */
   public double weight() ;
...
} // interface FoodProcessorEnvironmentIF
```

Напишем абстрактный класс, который является суперклассом для всех классов верхнего уровня программы:

```
public abstract class AbstractFoodProcessorProgram {
   private FoodProcessorEnvironmentIF environment;

   /**
    * Операционная среда пищевого процессора передает этому
```

```
     * методу ссылку на саму себя,
     * что позволяет экземплярам подклассов этого класса
     * вызывать методы объекта среды пищевого процессора,
     * реализующего интерфейс FoodProcessorEnvironmentIF.
     */
    public void setEnvironment(
              FoodProcessorEnvironmentIF environment) {
      this.environment = environment;
    } // setEnvironment(FoodProcessorEnvironmentIF)

    /**
     * Разрешает подклассам считать ссылку на операционную среду.
     */
    protected FoodProcessorEnvironmentIF getEnvironment() {
      return environment;
    } // getEnvironment()

    /**
     * Возвращает имя этого объекта программы приготовления пищи.
     */
    public abstract String getName() ;

    /**
     * При вызове этого метода программа приготовления
     * пищи получает сообщение о начале работы.
     */
    public abstract void start() ;
...
} // class AbstractFoodProcessorProgram
```

Приведем класс, который позволяет операционной среде пищевого процессора запускать программы. Он использует объект `ClassLoader` для управления загружаемыми им классами.

```
public class FoodProcessorEnvironment
              implements FoodProcessorEnvironmentIF {
  private static final URL[] classPath; // URL для программы.
  static {
    try {
      classPath = new URL[](new URL(«file:///bin»)};
    } catch (java.net.MalformedURLException e) {
```

```
            throw new ExceptionInInitializerError(e);
        } // try
    } // static

    /**
     * Нарезает пищу ломтиками заданной ширины.
     */
    public void slice(int width) {
...
    } // slice(int)

    /**
     * Смешивает пищу с заданной скоростью.
     */
    public void mix(int speed) {
...
    } // mix(int)

    /**
     * Взвешивает пищу.
     * Возвращает вес в унциях.
     */
    public double weigh() {
        double weight = 0.0;
...
        return weight;
    } // weight()
...
    /**
     * Запускает указанную программу.
     */
    void run(String programName) {
        // Создаем ClassLoader для загрузки классов программы.
        // Если эти классы больше не нужны,
        // то они могут быть удалены сборщиком мусора.
        URLClassLoader ClassLoader;
        ClassLoader = new URLClassLoader(classPath);
        Class programClass;
        try {
            programClass = ClassLoader.loadClass(programName);
```

```
        } catch (ClassNotFoundException e) {
        // Не найден.

        return;
        } // try
        AbstractFoodProcessorProgram program;
        try {
           program = (AbstractFoodProcessorProgram)
                programClass.newInstance();
        } catch (Exception e) {
           // Запуск невозможен.
           ...
           return;
        } // try
        program.setEnvironment(this);
        display(program.getName());
        program.start();
     } // run(String)
     ...
} // class FoodProcessorEnvironment
```

И наконец, пример кода для класса верхнего уровня программы:

```
public class ConcreteFoodProcessorProgram
        extends AbstractFoodProcessorProgram {
   /**
    * Возвращает имя этого объекта программы приготовления пищи.
    */
   public String getName() { return "Chocolate Milk"; }

   /**
    * Вызов этого метода сообщает программе приготовления
    * пищи о начале ее работы.
    */
   public void start() {
      double weight = getEnvironment().weigh();
      if (weight > 120.0 && weight < 160.0)
         getEnvironment().mix(4);
      ...
   } // start()
   ...
} // class ConcreteFoodProcessorProgram
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ DYNAMIC LINKAGE

Virtual Proxy. Реализация шаблона Virtual Proxy иногда использует шаблон Dynamic Linkage для загрузки класса, который нужен ему для создания своего основного объекта.

Protection Proxy. Шаблон Protection Proxy (описанный в книге [Grand2001]) иногда используется вместе с шаблоном Dynamic Linkage с целью минимизации риска, связанного с нарушением системы безопасности.

Virtual Proxy (Виртуальный заместитель)

Этот шаблон ранее был описан в работе [Larman98].

СИНОПСИС

Если инстанциирование объекта требует больших затрат, а объект может не потребоваться, то имеет смысл отложить его инстанциирование до тех пор, пока не станет действительно ясно, что объект нужен. Шаблон Virtual Proxy скрывает от своих клиентов тот факт, что объект еще может не существовать, предоставляя им косвенный доступ к объекту, осуществляемый через объект-заместитель, реализующий тот же интерфейс, что и объект, который может не существовать. В данном случае используется «ленивое» инстанциирование.

КОНТЕКСТ

Предположим, что группа программистов пишет объемный Java-апплет для компании, управляющей сетью больших хозяйственных магазинов. Этот апплет позволит людям покупать при помощи интернета все, что продается в этих магазинах. Помимо предложенного каталога апплет предусматривает использование различных помощников, позволяющих потребителям выбрать именно то, что им нужно. Эта помощь, например, включает:

- консультанта по кухонным шкафам, помогающего потребителю подобрать набор шкафов и затем автоматически заказать все детали, необходимые для сборки шкафов;
- консультанта, помогающего определить, сколько пиломатериалов потребуется покупателю для строительства настила;
- консультанта для определения размера коврового покрытия, соответствующего определенной планировке, и наилучшего способа разрезания этого покрытия.

Таких помощников на самом деле намного больше, и апплет получается очень объемный. Из-за своих размеров он требует от браузера недопустимо большого количества времени на загрузку при использовании модемного соединения.

Один из способов уменьшения времени загрузки апплета заключается в том, чтобы не загружать каких-либо консультантов до тех пор, пока они не потребуются. Шаблон Virtual Proxy предлагает способ задержки загрузки части апплета таким способом, который прозрачен для остальной части апплета. Идея заключается в том, что остальная часть апплета обращается к классам, входящим в состав консультанта, не прямо, а косвенно, через класс-заместитель. Классы-заместители специально создаются так, чтобы не содержать какой-либо

статической ссылки[1] на класс, заместителями которого они являются. Это означает, что, если классы-заместители загружены, нет ни одной ссылки на класс, для которого эти классы являются заместителями. Если остальная часть апплета ссылается только на заместителей, а не на классы, реализующие консультантов, то Java-машина не должна автоматически загружать консультантов.

При вызове метода заместителя он сначала обеспечивает загрузку и инстанциирование классов, реализующих консультанта. Затем он вызывает соответствующий метод через интерфейс.

На рис. 7.23 изображена основная часть апплета, которая ссылается на класс `CabinetAssistantProxy`, реализующий интерфейс `CabinetAssistantIF`. Основная часть апплета не содержит ссылок на классы, реализующие консультанта по шкафам. В случае необходимости класс `CabinetAssistantProxy` обеспечивает загрузку и инстанциирование классов, реализующих консультанта по шкафам. Код, реализующий этот механизм, приводится в разделе «Пример кода».

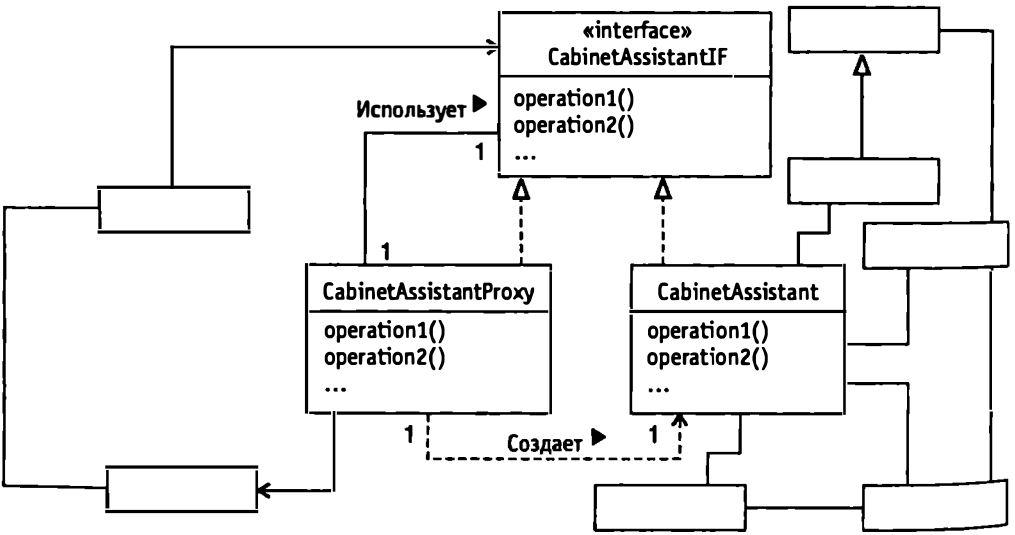


Рис. 7.23. Класс-заместитель консультанта по шкафам

---

[1] Под статической ссылкой автор понимает ссылку на класс, который компилятор будет распознавать на стадии компиляции. Например,

```
Foo myFoo;
```

это статическая ссылка на класс `Foo`. Сравните со статической ссылкой на `Foo` из следующего примера:

```
Class clazz = Class.forName("Foo");
```

В этом примере компьютер видит строку, которая содержит имя класса. Строка не распознается как имя класса до тех пор, пока не начнется стадия выполнения и не будет вызван метод `forName`.

МОТИВЫ

☺ Инстанциирование класса требует много времени.

☺ Инстанциирование класса может не потребоваться.

☺ Если существует несколько классов, экземпляры которых не потребуются в течение неопределенного промежутка времени, то инстанциирование их всех одновременно может привести к значительной задержке работы программы. Задержка их инстанциирования до момента, пока они не понадобятся на самом деле, позволяет распределить время, затрачиваемое программой на их инстанциирование, что способствует улучшению программы.

☺ Управление отложенным инстанциированием классов не должно быть возложено на клиентов класса. Поэтому отложенное инстанциирование класса должно быть прозрачным для его клиентов.

☹ Иногда самый лучший способ обеспечить хорошую производительность программы состоит в том, чтобы продолжить ее инициализацию таким образом, чтобы все объекты, инстанциирование которых требует больших затрат, создавались бы при запуске программы. Тогда, возможно, не придется позже затрачивать время на инициализацию.

РЕШЕНИЕ

На рис. 7.24 показана организация классов, которые участвуют в шаблоне Virtual Proxy.

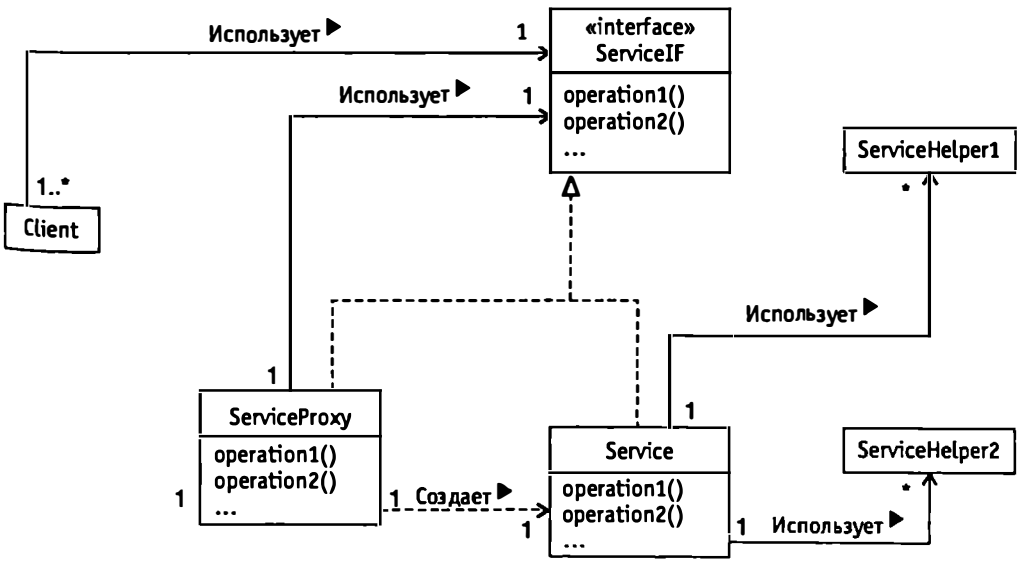


Рис. 7.24. Шаблон Virtual Proxy

Опишем роли, исполняемые этим интерфейсом и классами.

Service. Класс `Service` обеспечивает логику верхнего уровня для предоставляемого им сервиса. При создании экземпляра этого класса класс создает другие объекты, которые ему нужны. Такие классы указаны на диаграмме как `ServiceHelper1`, `ServiceHelper2`.

Client. Класс `Client` использует сервис, предоставляемый классом `Service`. Классы `Client` никогда прямо не используют класс `Service`. Вместо этого они используют класс `ServiceProxy`, обеспечивающий функциональность класса `Service`. Косвенное использование класса `Service` делает клиентские классы нечувствительными к тому, существует ли уже экземпляр класса `Service`, косвенно используемый объектами `Client`.

ServiceProxy. Задача класса `ServiceProxy` состоит в задержке создания экземпляров класса `Service` до тех пор, пока они действительно не понадобятся.

Класс `ServiceProxy` обеспечивает косвенность между классами `Client` и классом `Service`. Косвенность скрывает от объектов `Client` тот факт, что если создан объект `ServiceProxy`, то соответствующий объект `Service` не существует, а класс `Service` может даже и не быть загружен.

Объект `ServiceProxy` отвечает за создание соответствующего объекта `Service` и создает объект `Service` при первом же запросе на выполнение операции, требующей наличия объекта `Service`.

Класс `ServiceProxy` пишется специально для получения доступа к классу `Service` через динамическую ссылку. Обычно классы ссылаются на другие классы при помощи статических ссылок. Статическая ссылка просто содержит имя класса, который находится в соответствующем месте исходного кода. Когда компилятор видит ссылку такого рода, он автоматически загружает другой класс вместе с классом, содержащим ссылку на него.

Шаблон Virtual Proxy препятствует загрузке класса `Service` и связанных с ним классов вместе с остальной частью программы, гарантируя, что эта остальная часть программы не содержит каких-либо статических ссылок на класс `Service`. Вместо этого остальная часть программы ссылается на класс `Service` через класс `ServiceProxy`, а класс `ServiceProxy` ссылается на класс `Service` при помощи динамической ссылки.

Динамическая ссылка содержит вызов метода, передающего строку с именем класса методу, загружающему этот класс, если он еще не загружен, и возвращает ссылку на этот класс. Как правило, в таких случаях вызывается статический метод `java.lang.class.forName`. Поскольку имя класса может находиться только внутри строки, то компиляторы не знают, на какой класс нужно будет ссылаться, и поэтому они не генерируют сигнал, заставляющий класс загружаться.

ServiceIF. Класс `ServiceProxy` создает экземпляр класса `Service`, вызывая методы, не требующие каких-либо статических ссылок на класс `Service`. Класс `ServiceProxy` тоже обращается к методам класса `Service`, не имея каких-либо статических ссылок на класс `Service`. Он вызывает методы класса `Service` благодаря тому, что класс `Service` реализует интерфейс `ServiceIF`.

Интерфейс ServiceIF объявляет все методы, реализуемые классом Service и необходимые для класса ServiceProxy. Поэтому объект ServiceProxy может рассматривать создаваемую им ссылку на объект Service как ссылку на объект ServiceIF. Класс Service использует статические ссылки на интерфейс ServiceIF для вызова методов объектов Service. Никакие статические ссылки на класс Service не нужны.

РЕАЛИЗАЦИЯ

Совместно используемые объекты сервиса

Согласно этому решению, когда объекту ServiceProxy впервые поступает запрос на выполнение некоторой операции, он создает объект Service и впоследствии постоянно будет иметь ассоциируемый с ним объект Service. Но если объект Service требует на время своего существования большого объема памяти или много разных ресурсов, то нецелесообразно создавать столько объектов Service, сколько имеется объектов ServiceProxy.

Если объекты Service не имеют состояний и взаимозаменяемы, то можно рассмотреть возможность использования шаблона Object Pool с целью сведения к минимуму количества объектов Service, которые создает разработчик. Идея заключается в том, что, когда объекту ServiceProxy для выполнения некоторой операции нужен объект Service, он получает объект Service из пула объектов. Когда объект Service заканчивает выполнение требуемой операции, объект ServiceProxy возвращает его назад в пул объектов. Этот способ позволяет иметь множество объектов ServiceProxy и всего лишь несколько объектов Service.

Отложенная загрузка класса

Во многих случаях класс, доступный через виртуальный заместитель, применяет другие классы, которые не используются остальной частью программы. При таких отношениях эти классы не загружаются до тех пор, пока не будет загружен класс, доступный через виртуальный заместитель. Если важно, чтобы такие классы не загружались, пока не будет загружен класс, доступный через виртуальный заместитель, то проблема может возникнуть на этапе имплементации. Программист может добавить прямую ссылку на один из таких классов, не задумываясь о последствиях. Если тесты программы, проверяющие ее качество, не включают тестирование производительности, проблема может остаться незамеченной до тех пор, пока не поступят жалобы от пользователей этой программы.

Можно уменьшить вероятность такого хода событий, сделав отношение между классами явным. Для этого можно поместить упомянутые классы в пакет, причем видимым за пределами пакета может быть только один используемый заместителем класс (рис. 7.25).

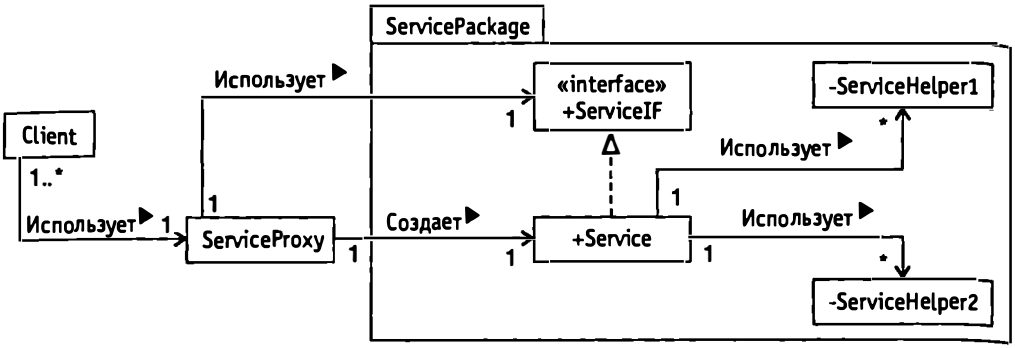


Рис. 7.25. Отношение становится явным при использовании пакета

СЛЕДСТВИЯ

☺ Классы, к которым остальная часть программы обращается исключительно через виртуальный заместитель, не загружаются до тех пор, пока не будут нужны.

☺ Объекты, доступные через виртуальный заместитель, не создаются до тех пор, пока они не понадобятся.

☺ Классы, использующие заместитель, не нуждаются в сведениях о том, загружен ли класс `Service`, существует ли его экземпляр или даже существует ли сам класс.

• Все классы, не являющиеся классами заместителей, должны осуществлять доступ к предоставляемому классом `Service` сервису косвенно, через заместителя. Это очень важно. Если хотя бы один класс обращается к классу `Service` напрямую, то класс `Service` будет загружаться до того, как он потребуется на самом деле. В этом заключена скрытая ошибка, которая обычно влияет на производительность, а не на саму функциональность, и поэтому ее очень трудно отследить.

ПРИМЕР КОДА

Чтобы завершить пример, начатый в разделе «Контекст», приведем код, реализующий консультанта по шкафам и класс заместителя. Сначала — соответствующий код для класса `CabinetAssistant`:

```
/**
* Это класс сервиса, используемый виртуальным заместителем.
* Класс реализует интерфейс, написанный исключительно
* для объявления методов этого класса.
```

```
*/
public class CabinetAssistant implements CabinetAssistantIF (
   public CabinetAssistant(String s) (
...
   } // Constructor(String)
...
   public void operation1() {
...
   } // operation1()

   public void operation2() {
...
   } // operation2()
} // class CabinetAssistant
```

Интерфейс `CabinetAssistantIF` просто объявляет методы, определенные классом `CabinetAssistant`:

```
public interface CabinetAssistantIF {
   public void operation1();
   public void operation2();
...
} // interface CabinetAssistantIF
```

И наконец, приведем код для класса `CabinetAssistantProxy`, где и происходит все самое интересное:

```
public class CabinetAssistantProxy
             implements CabinetAssistantIF {
   private CabinetAssistantIF assistant = null;

   // Для конструктора объекта консультанта.
   private String myParam;

   public CabinetAssistantProxy(String s) {
      myParam = s;
   } // constructor(String)

   /**
    * Получаем объект CabinetAssistant, который используется
    * для реализации операций. Если он еще не существует,
    * этот метод создает его.
    */
```

```
    private CabinetAssistantIF getCabinetAssistant() {
      if (assistant == null) {
        try {
          // Получаем объект класса,
          // который представляет класс Assistant.
        Class clazz;
          clazz = Class.forName("CabinetAssistant");

          // Получаем объект конструктора для доступа
          // к конструктору класса CabinetAssistant,
          // принимающему единственный строковый аргумент.
          Constructor constructor;

          // Получаем объект конструктора
          // для создания объекта CabinetAssistant.
          Class[] formalArgs;
          params = new Class [] { String.class };
          constructor = clazz.getConstructor(params);

          // Используем объект конструктора.
          Object[] actuals = new Object[] { myParam };
          Assistant = (CabinetAssistantIF)
            constructor.newInstance(actuals);
        } catch (Exception e) {
        } // try
        if (assistant == null) {
            // Обработка случая неуспешного создания
            // объекта CabinetAssistant.
          throw new RuntimeException();
        } // if
      } // if
      return assistant;
    } // getCabinetAssistant()

    public void operation1() {
      getCabinetAssistant().operation1();
    } // operation1()

    public void operation2() {
      getCabinetAssistant().operation2();
    } // operation2()
...
} // class CabinetAssistantProxy
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ VIRTUAL PROXY

Facade. Шаблон Facade может использоваться вместе с шаблоном Virtual Proxy для того, чтобы количество необходимых классов-заместителей было минимальным.

Proxy. Шаблон Virtual Proxy — это специальная версия шаблона Proxy.

Object Pool. Шаблон Object Pool можно использовать с той целью, чтобы позволить многочисленным объектам `ServiceProxy` совместно использовать всего лишь несколько объектов `Service`.

Decorator (Декоратор)

Шаблон Decorator известен также как шаблон Wrapper. Он был ранее описан в работе [GoF95].

СИНОПСИС

Шаблон Decorator расширяет функциональные возможности объекта, используя прозрачный для его клиентов способ: он реализует тот же самый интерфейс, что и исходный класс, и делегирует исходному классу операции.

КОНТЕКСТ

Предположим, необходимо разработать ПО для системы безопасности, которая контролирует физический доступ в здание. Его основная архитектура такова, что устройство считывания карточек или другое устройство ввода данных получает некоторую идентификационную информацию и передает эту информацию объекту, контролирующему дверь. Если контролирующий дверь объект удовлетворен информацией, он разблокирует дверь. На рис. 7.26 представлена диаграмма взаимодействия, описывающая этот механизм.

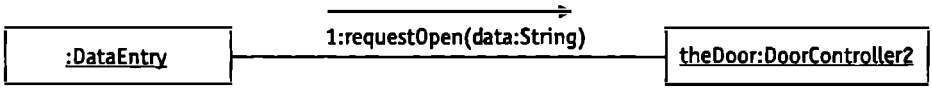


Рис. 7.26. Основы контролирования физического доступа

Допустим, нужно объединить этот механизм контроля доступа с системой наблюдения. Обычно в системе наблюдения намного больше подсоединенных к ней камер, чем телевизионных мониторов. Большинство мониторов периодически показывают изображения, полученные от различных камер, демонстрируя картинку от одной камеры в течение нескольких секунд и затем переходя к следующей камере, относящейся к этому монитору. Существуют некоторые правила установки системы наблюдения, гарантирующие ее эффективность. В данном случае рассмотрим следующие правила:

- по крайней мере, одна камера следит за каждой входной дверью, связанной с системой контроля доступа;
- каждый монитор отвечает только за одну камеру, следящую за входной дверью с контролируемым доступом. Это объясняется тем, что если за входной дверью будет следить несколько камер, то поломка одного монитора не помешает просмотру изображений от всех камер на входной двери.

Специальное требование интеграции заключается в том, что если контролирующий дверь объект получает запрос открыть дверь, то мониторы, отвечающие за камеры, направленные на эту входную дверь, показывают ее. Чтобы удовлетворить это требование, можно расширить класс или написать несколько подклассов (рис. 7.27).

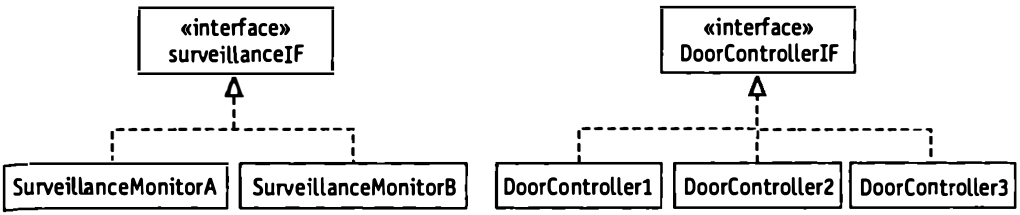


Рис. 7.27. Классы системы безопасности

Установленные двери могут быть трех видов, а применяемые мониторы наблюдения могут соответствовать двум разным типам. Можно решить проблему, написав два подкласса для каждого из классов контроллеров дверей. Однако самое лучшее — не писать шесть классов, а использовать шаблон Decorator, который решает эту задачу посредством делегирования, а не наследования.

Нужно написать два новых класса: `DoorControllerWrapperA` и `DoorControllerWrapperB`. Оба эти класса реализуют интерфейс `DoorControllerIF` (рис. 7.28). Они наследуют реализацию интерфейса `DoorControllerIF` от своего абстрактного суперкласса `AbstractDoorControllerWrapper`.

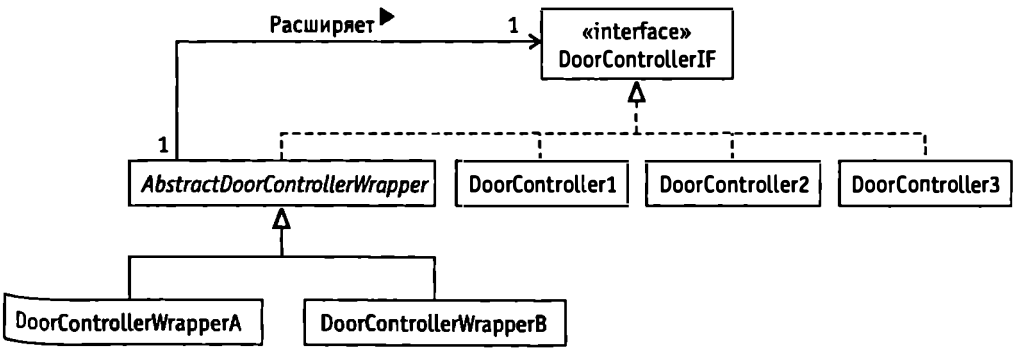


Рис. 7.28. Классы контроллера двери

Класс `AbstractDoorControllerWrapper` реализует все методы интерфейса `DoorControllerIF`, просто вызывая соответствующий метод другого объекта, реализующего интерфейс `DoorControllerIF`. Классы `DoorControllerA` и `DoorControllerB` представляют собой классы-декораторы, не являющиеся абстрактными. Они расширяют возможности поведения реализации `requestOpen`,

унаследованного ими также для выдачи запроса монитору на отображение картинки обзора этой двери (рис. 7.29).

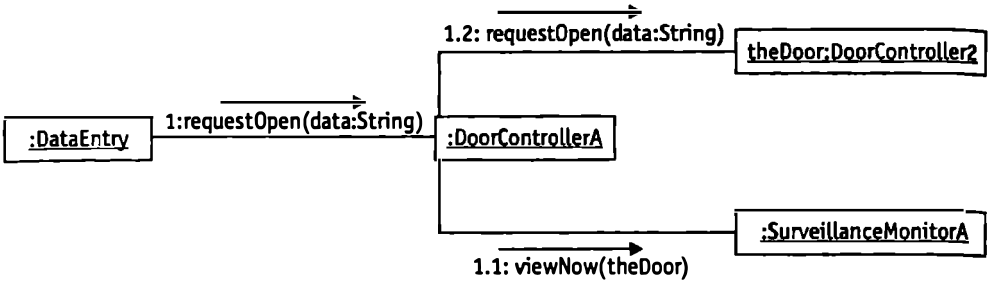


Рис. 7.29. Взаимодействие при наблюдении за дверью

Такой подход позволяет просматривать входную дверь при помощи нескольких камер, которые должны управляться посредством простого размещения нескольких классов-декораторов перед объектом `DoorControllerIF`.

МОТИВЫ

☺ Нужно расширить функциональные возможности класса, однако существуют причины, не позволяющие выполнить расширение при помощи наследования.

☺ Существует необходимость в динамическом расширении функциональных возможностей объекта.

РЕШЕНИЕ

На рис. 7.30 представлена диаграмма классов, на которой изображена основная структура шаблона Decorator.

Опишем роли, исполняемые классами и интерфейсом в шаблоне Decorator.

AbstractServiceIF. Интерфейс, выступающий в этой роли, реализуется всеми объектами сервиса, которые потенциально могут быть расширены с помощью шаблона Decorator. Классы, экземпляры которых могут быть использованы для динамически расширенных классов, реализующих интерфейс `AbstractServiceIF`, тоже должны реализовывать интерфейс `AbstractServiceIF`.

ConcreteService. Классы, играющие эту роль, обеспечивают основную функциональность, которая расширяется благодаря шаблону Decorator.

AbstractWrapper. Абстрактный класс, выступающий в этой роли, — это общий суперкласс для классов-оберток. Экземпляры этого класса содержат также ссылку на объект `AbstractServiceIF`, которому объекты `ConcreteWrapper` делегируют операции.

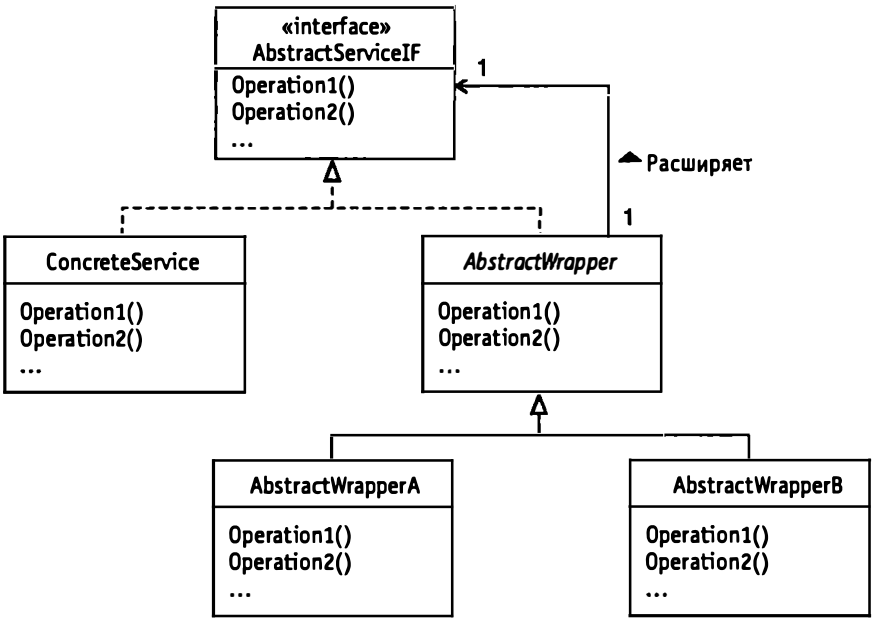


Рис. 7.30. Шаблон Decorator

Кроме того, этот класс обычно реализует все методы, объявленные интерфейсом `AbstractServiceIF`, поэтому они просто вызывают имеющий похожее имя метод объекта `AbstractServiceIF`, которому объект-обертка делегирует операции. Тем самым задается реализация по умолчанию для методов, функциональность которых не изменяется.

ConcreteWrapperA, ConcreteWrapperB и т.д. Эти конкретные классы-обертки расширяют возможности поведения методов, унаследованного ими от класса `AbstractWrapper`, любым приемлемым способом.

РЕАЛИЗАЦИЯ

Большая часть реализаций шаблона Decorator проще, чем общий случай. Приведем некоторые общие упрощенные варианты.

- Если существует только один класс `ConcreteService` и нет класса `AbstractService`, то `AbstractWrapper` может быть подклассом класса `ConcreteService`.
- Если существует только один конкретный класс-обертка, нет необходимости в существовании отдельного класса `AbstractWrapper`. Можно объединить обязанности класса `AbstractWrapper` с обязанностями конкретного класса-обертки. Подобный отказ от класса `AbstractWrapper` может быть уместным также в том случае, когда имеются два конкретных класса-обертки, но не более того.

СЛЕДСТВИЯ

☺ Шаблон Decorator предоставляет большую гибкость, чем наследование. Он позволяет динамически изменять поведение отдельных объектов, добавляя и убирая классы-обертки. С другой стороны, наследование определяет суть всех экземпляров класса статическим образом.

☺ Используя различные комбинации декораторов нескольких видов, можно задавать многочисленные комбинации поведения. Чтобы создать множество различных видов поведения при помощи наследования, потребуется определение множества различных классов.

• Как правило, при использовании шаблона Decorator получается меньше классов, чем при использовании наследования. Меньшее количество классов упрощает проектирование и реализацию программ. С другой стороны, использование шаблона Decorator обычно приводит к большему количеству объектов.

☹ Из-за своей гибкости объекты декораторов более подвержены ошибкам, чем унаследованные объекты. Например, существует вероятность такой комбинации объектов-оберток, при которой они не будут работать, или возможно создание циклических ссылок между объектами декораторов.

☹ Шаблон Decorator затрудняет использование уникальности объектов для идентификации объектов сервиса, поскольку скрывает объекты сервиса за объектами-декораторами.

ПРИМЕР КОДА

Приведем код, который реализует некоторые классы контроллеров дверей, представленные на диаграммах в разделе «Контекст». Реализация интерфейса `DoorControllerIF` выглядит так:

```
interface DoorControllerIF {
   /**
    * Запрашивает открытие двери, если заданный ключ подходит.
    */
   public void requestOpen(String key);

   /**
      * Закрывает дверь.
      */
   public void close();
   ...
} // interface DoorControllerIF
```

А теперь — класс AbstractDoorControllerWrapper, который обеспечивает для своих подклассов задаваемые по умолчанию реализации методов, объявленных интерфейсом DoorControllerIF:

```
abstract class AbstractDoorControllerWrapper
              implements DoorControllerIF {
  private DoorControllerIF wrappee;
  /**
   * Constructor
   * @param wrappee Объект, которому этот объект будет
   * делегировать операции.
   */
  AbstractDoorControllerWrapper(DoorControllerIF wrappee) {
    this.wrappee = wrappee;
  } // constructor(wrappee)

  /**
   * Запрашивает открытие двери, если данный ключ подходит.
   */
  public void requestOpen(String key) {
    wrappee.requestOpen(key);
  } // requestOpen(String)

  /**
   * Закрывает дверь.
   */
  public void close() {
    wrappee.close();
  } // close()
...
} // class AbstractDoorControllerWrapper
```

И наконец, подкласс класса AbstractDoorControllerWrapper, который расширяет заданное по умолчанию поведение, запрашивая монитор с целью отображения картинки от указанной камеры:

```
class DoorControllerWrapperA
              extends AbstractDoorControllerWrapper {
  private String camera;  // Задаем имя камеры, просматривающей
                          // эту дверь.
  private SurveillanceMonitorIF monitor; // Монитор этой камеры.
```

```
    /**
     * Constructor
     * @param wrappee Объект DoorController, которому этот объект
     *                будет делегировать операции.
     * @param camera Идентификатор камеры, просматривающей эту
     *               дверь.
     * @param monitor Монитор, запрашиваемый для просмотра
     *                картинки от камеры.
     */
    DoorControllerWrapperA(DoorControllerIF wrappee,
                           String camera,
                           SurveillanceMonitorIF monitor) {
      super(wrappee);
      this.camera = camera;
      this.monitor = monitor;
    } // constructor(wrappee)

    /**
     * Запрашивает открытие двери, если данный ключ подходит.
     */
    public void requestOpen(String key) {
      monitor.viewNow(camera);
      super.requestOpen(key);
    } // requestOpen(String)
  } // class DoorControllerWrapperA
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ DECORATOR

Delegation. Шаблон Decorator представляет собой структурный способ применения шаблона Delegation.

Filter. Шаблон Filter — это специальная версия шаблона Decorator, предназначенная для обработки потока данных.

Strategy. Шаблон Decorator полезен для дополнительных действий до или после обращения к методам другого объекта. Если необходимы какие-то действия во время обращения к методу, рекомендуем рассмотреть использование шаблона Strategy.

Template Method. Template Method — это альтернативный вариант шаблона Decorator, который разрешает вариации поведения не до или после вызова метода, а во время обращения к методу.

Cache Management (Управление кэшем)

СИНОПСИС

Шаблон Cache Management позволяет осуществить быстрый доступ к объектам, обращение к которым в противном случае потребовало бы значительного времени. Он хранит копию тех объектов, создание которых требует больших затрат. Создание объекта может потребовать больших затрат в силу множества причин, например, выполнения длительных вычислений или считывания информации из базы данных.

КОНТЕКСТ

Предположим, создается программа, которая позволит потребителям читать информацию об изделиях в каталоге. Считывание всех данных об изделии может занимать несколько секунд, поскольку иногда информация может поступать от многих источников. Сохранение информации об изделии в памяти программы может ускорить события при следующем запросе информации, поскольку не нужно будет тратить время на сбор этой информации.

Если выборка информации требует достаточно много времени, то эта информация может сохраняться в памяти с целью быстрого доступа к ней в случае необходимости; такая технология называется *кэшированием*. Поскольку каталог может описывать очень большое количество продуктов, невозможно сохранить в кэш-памяти информацию обо всех изделиях. Но можно сделать следующее: сохранить информацию о таком количестве изделий, какое позволяет память. В памяти должна содержаться информация о тех изделиях, которые, как полагают, будут использоваться чаще всего. Информация об изделиях, которые, согласно предположениям, будут менее используемыми, не будет помещаться в память. Принятие решения, какие объекты и в каком количестве должны быть сохранены в памяти, называется *управлением кэшем*.

Рис. 7.31 демонстрирует, каким образом осуществляется управление кэшем при получении информации об изделиях:

1. Идентификатор (ID) изделия передается методу `getProductInfo` объекта `ProductCacheManager`.

   1.1. Метод `getProductInfo` объекта `ProductCacheManager` пытается получить объект описания изделия из объекта `Cache`. При успешном получении информации из кэша он возвращает объект описания изделия.

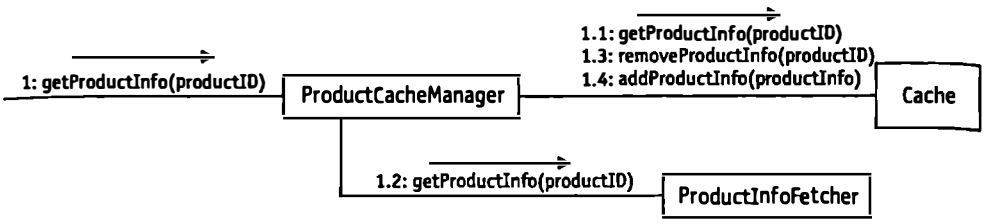


Рис. 7.31. Взаимодействие при управлении кэшем изделий

1.2. Если метод `getProductInfo` не может прочитать информацию из кэша, то для получения объекта описания изделия из кэша он вызывает метод `getProductInfo` объекта `ProductInfoFetcher`.

1.3. Кэш-менеджеры обычно ограничивают количество объектов в кэше, так как сохранение большого количества объектов может привести к излишней расточительности или даже снизить производительность. Если кэш-менеджер принимает решение, что считанный объект должен быть сохранен в кэше, но кэш уже заполнен, кэш-менеджер не будет увеличивать количество объектов в кэше. Он выберет объект описания изделия, который должен быть удален из кэша, и передаст идентификатор этого изделия методу `removeProductInfo` объекта `Cache`.

1.4. И наконец, если кэш-менеджер решил, что считанный объект описания изделия должен быть сохранен в кэше, он вызывает метод `addProductInfo` объекта `Cache`.

МОТИВЫ

☺ Необходим доступ к объекту, создание или считывание которого требует длительного времени. Создание объекта обычно требует больших затрат, если нужно считывать его содержимое из внешних источников или выполнять длительные вычисления. Суть в том, что создание объекта требует намного больше времени, чем доступ к этому объекту, сохраненному во внутренней памяти.

☺ Если количество объектов, создание которых требует больших затрат, достаточно невелико, и все они могут свободно поместиться в локальной памяти, то сохранение всех этих объектов в локальной памяти обеспечит наилучший результат. Это гарантирует, что, если к одному из таких объектов снова потребуется доступ, не надо будет затрачивать усилия на повторное создание объекта.

☺ Если создается множество объектов и их построение требует больших затрат, то все они могут не поместиться в памяти одновременно. Даже если все они могут поместиться, занимаемая ими память может потребоваться

для других целей. Поэтому иногда необходимо установить верхний предел количества объектов, находящихся в кэше локальной памяти.

☺ Установление верхнего предела, ограничивающего количество находящихся в кэше объектов, предполагает проведение политики принуждения, в соответствии с которой должно приниматься решение, какие прочитанные объекты поместить в кэш, а какие отбросить в том случае, если размеры кэша достигли верхней границы. Если придерживаться этой политики, нужно попытаться предсказать, какие объекты наиболее и наименее вероятно будут использоваться в ближайшем будущем.

☹ Некоторые объекты отражают состояние чего-то, что находится вне собственной памяти программы. Содержимое таких объектов может стать неправильным спустя некоторое время с момента создания объекта.

РЕШЕНИЕ

На рис. 7.32 представлена общая структура шаблона Cache Management.

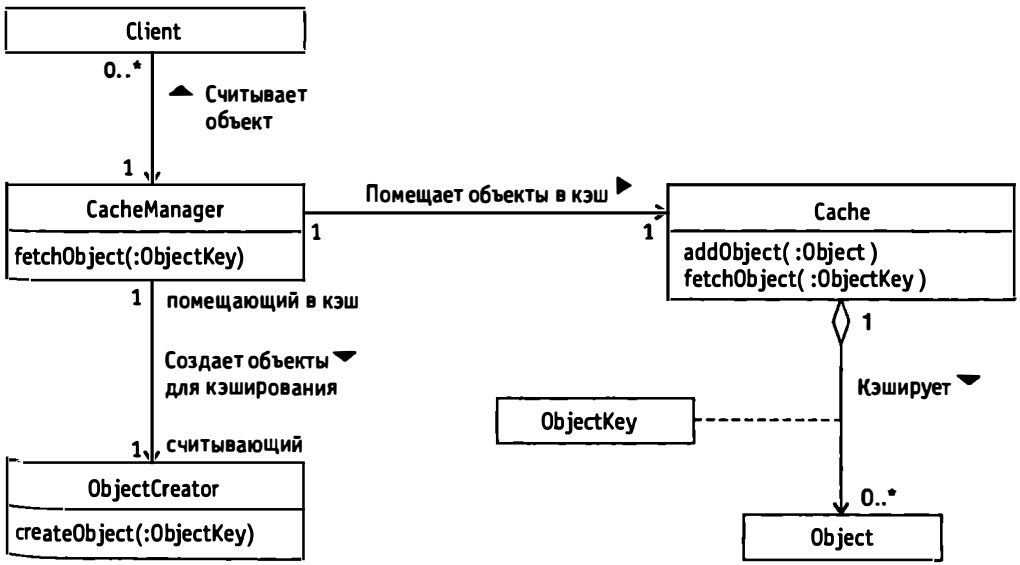


Рис. 7.32. Шаблон Cache Management

Опишем классы, принимающие участие в этом шаблоне, и роли, которые они исполняют.

Client. Экземпляры классов, выступающих в этой роли, делегируют ответственность по получению доступа к заданным объектам объекту `CacheManager`.

ObjectKey. Экземпляры класса `ObjectKey` идентифицируют считываемый или создаваемый объект.

CacheManager. Объекты `Client` запрашивают объекты у объекта `CacheManager`, вызывая его метод `fetchObject`. Аргументом метода `fetchObject` служит объект `ObjectKey`, который идентифицирует считываемый объект. Сначала метод `fetchObject` вызывает метод `fetchObject` объекта `Cache`. В случае неудачи он вызывает метод `createObject` объекта `ObjectCreator`.

ObjectCreator. Объекты `ObjectCreator` отвечают за создание объектов, которые не находятся в кэше.

Cache. Объект `Cache` отвечает за управление коллекцией объектов в кэше. Если задан объект `ObjectKey`, объект `Cache` быстро находит соответствующий объект, содержащийся в кэше. Объект `CacheManager`, чтобы получить объект из кэша, передает объект `ObjectKey` методу `fetchObject` объекта `Cache`. Если метод `fetchObject` не возвращает нужный объект, объект `CacheManager` просит объект `ObjectCreator` создать этот объект. Если объект `ObjectCreator` возвращает затребованный объект, объект `Cache` передает полученный объект методу `addObject` этого объекта. Метод `addObject` добавляет объект в кэш, если это согласуется с его политикой управления кэшем. Метод `addObject` может удалить объект из кэша, чтобы освободить место для объекта, добавляемого в кэш.

РЕАЛИЗАЦИЯ

Описание структуры

При проектировании обоих классов: и `CacheManager`, и `ObjectCreator` — следует учесть то, что они должны реализовывать общий интерфейс. Объекты `Client` должны получать доступ к объектам `CacheManager` через общий интерфейс, что делает использование кэша прозрачным для объектов `Client`. Если они реализуют общий интерфейс, то объекты `Client` используют объект, который реализует тот же самый интерфейс, независимо от того, применяется ли кэширование или нет.

Этот интерфейс не упоминается в разделе «Решение» как часть данного шаблона, поскольку классы, выступающие в роли `ObjectCreator`, часто проектируются до того, как начинает рассматриваться кэширование. Поэтому объекты `ObjectCreator` часто не реализуют какой-либо подходящий интерфейс.

Реализация кэша

Реализация шаблона Cache Management предусматривает принятие некоторых потенциально сложных решений. Оптимальный вариант может быть выбран на основе обширного статистического анализа, теории массового обслуживания и других видов математического анализа. Однако обычно имеется возможность осуществлять приемлемую реализацию, используя информацию о готовых вариантах и экспериментируя с различными решениями.

При реализации шаблона Cache Management основное решение заключается в том, как реализовать сам кэш. Приведем некоторые соображения по поводу выбора структуры данных для кэша:

- кэш должен уметь быстро находить объекты по заданному для них `ObjectKey`;
- поиск будет производиться чаще добавления или удаления, поэтому он должен выполняться так же быстро или быстрее добавления или удаления;
- предполагается частое использование операций добавления и удаления объектов, поэтому структура данных не должна рассматривать эти операции как требующие намного больше затрат, чем операции поиска.

Этим требованиям удовлетворяет хэш-таблица. Если кэш реализуется на языке Java, то он обычно создается при помощи экземпляра класса `java.util.HashMap` или `java.util.Hashtable`.

Настройка производительности кэша

Остальные вопросы реализации относятся к настройке производительности. Этой теме не следует уделять внимание до тех пор, пока программа не начнет правильно функционировать. На этапах проектирования и начального кодирования следует задать первоначальные решения в отношении рассматриваемых вопросов, а затем просто не обращать на них внимания до тех пор, пока вы не будете готовы рассматривать вопросы, связанные с производительностью.

Самый простой способ оценки эффективности кэширования состоит в том, чтобы вычислить некоторую статистическую величину, которая называется *частотой успешных обращений* (hit rate). Частота успешных обращений представляет собой процент запросов на считывание объектов, которые были удовлетворены кэш-менеджером, предоставившим объекты, хранящиеся в кэше. Если все запросы были удовлетворены с помощью объекта, находящегося в кэше, то частота успешных обращений составляет 100 %. Если ни один запрос не был удовлетворен подобным образом, частота успешных обращений равняется 0 %. Частота успешных обращений в значительной степени зависит от того, насколько хорошо реализация шаблона Cache Management соответствует принципу запрашивания объектов.

Всегда существует максимальный размер памяти, которую можно выделить для кэша. Это означает, что нужно задать предельное количество объектов, которые могут находиться в кэше. Если предполагаемое количество таких объектов невелико, то не следует задавать явный предел. Но большинство проблем не имеет такого простого решения.

Предварительно задать максимальные размеры отводимой для кэша памяти не так легко, поскольку можно не знать заранее, сколько будет свободной памяти или сколько памяти потребуется для остальной части программы. Задать предельные размеры памяти, выделяемой для кэша, особенно проблематично в языке Java, поскольку не существует определенного соотношения между объектом и размером физической памяти, которую он занимает.

Альтернативой заданию и установлению предельных размеров памяти может служить простой подсчет объектов, который производится очень легко, поэтому можно упростить задачу, ограничивая содержимое кэша определенным количеством объектов.

Конечно, если существует ограничение на размеры кэша, то неизбежен вопрос: что случится, когда размеры кэша будут соответствовать максимальному разрешенному количеству объектов и при этом создается новый объект. С этого момента кэш должен хранить на один объект больше, чем предполагалось. И кэш-менеджер должен отбросить некий объект. Выбор удаляемого из кэша объекта очень важен, так как он прямо воздействует на частоту успешных обращений. Если отбрасываемый объект всегда будет следующим запрашиваемым объектом, то частота успешных обращений будет равна 0 %. С другой стороны, если отброшенный объект не потребуется раньше всех остальных, находящихся в кэше объектов, то отбрасывание этого объекта окажет минимальное негативное воздействие на частоту успешных обращений. Очевидно, что правильный выбор отбрасываемого объекта требует прогнозирования будущих запросов объектов.

В некоторых случаях, зная область применения, можно сделать разумное предположение по поводу того, какие объекты потребуются программе в ближайшем будущем. В наиболее благоприятных ситуациях можно с высокой степенью вероятности предсказать, какой именно объект будет затребован следующим. В таких случаях, если объект еще не находится в кэше, то он может быть срочно асинхронно создан в первую очередь, не дожидаясь запроса к нему со стороны программы. Это называется *выборкой объекта с упреждением.*

В большинстве случаев область применения не предоставляет достаточной информации для выполнения таких точных предсказаний. Однако существует шаблон, встречающийся так часто, что он стал основой эффективной задаваемой по умолчанию стратегии принятия решения об отбрасывании объекта. Эта стратегия основывается на том, что чем меньше прошло времени с тех пор, как программа запрашивала некоторый объект, тем больше вероятность, что она запросит его снова. Таким образом, всегда отбрасывается объект кэша, менее всех использовавшийся за последнее время. Эту стратегию часто сокращенно называют LRU (Least Recently Used, использовавшийся наиболее давно).

А теперь рассмотрим задание числового предела для находящихся в кэше объектов. Математический анализ может предоставить точное значение для максимального количества объектов, помещаемых в кэш. Но подобный анализ обычно не применяется по двум причинам. Первая состоит в том, что математический анализ предполагает использование теорий вероятностей и массового обслуживания, о которых большинство программистов ничего не знают. Другая причина заключается в том, что такой анализ может потребовать недопустимо много времени. Должно быть собрано очень много информации о программе и ее операционной среде. Тем не менее приемлемые размеры кэша обычно можно вычислить эмпирическим путем.

Сначала добавим код в класс `CacheManager` с целью оценки частоты успешных обращений, рассматриваемой как отношение количества запросов объектов,

удовлетворенных при помощи кэша, к общему количеству запросов объектов. Затем можно попробовать поработать с различными предельными размерами объекта. Выполнив эти действия, можно заметить, что, если кэш слишком велик, это может привести к замедлению или полному прекращению работы остальной части программы. Программа может не работать из-за нехватки памяти.

Предположим, что нужно настроить программу, использующую кэш. Запускают программу в идентичных условиях, задавая различные максимальные размеры кэша. Допустим, задают значение 6000 объектов и отмечают, что при 6000 на выполнение программы требуется примерно в три раза больше времени, чем при 4000. Это значит, что 6000 — слишком большое значение. В табл. 7.1 указаны возможные значения частоты успешных обращений, которые можно получить для разных значений размеров кэша.

Таблица 7.1. Размеры кэша и частота успешных обращений

Максимальный размер кэша, объектов	Частота успешных обращений, %
250	20
500	60
1000	80
2000	90
3000	98
4000	100
5000	100

Очевидно, что нет необходимости делать размер кэша больше, чем на 4000 объектов, поскольку уже при этом значении частота удачных обращений достигает 100 %. В данных условиях запуска программы идеальный размер кэша равен 4000 объектам. Если программа будет работать точно в таких условиях, то дальнейшая настройка может не понадобиться. Если программа будет работать в других условиях, можно использовать кэш меньших размеров, чтобы избежать проблем в случае меньших размеров свободной памяти. Выбранное число должно представлять собой компромисс между предполагаемой высокой частотой успешных обращений и желательно небольшим размером кэша. Поскольку уменьшение размеров кэша до 3000 объектов снижает частоту успешных обращений только до 98 %, то значение 3000 может отражать приемлемый размер кэша. А если приемлемая частота успешных обращений равна 90 %, то примерный размер кэша составляет 2000.

Если невозможно достичь большой частоты успешных обращений при приемлемых размерах памяти, а создание объектов требует больших затрат, то нужно рассмотреть вопрос об использовании вторичного кэша. Обычно вторичный кэш представляет собой файл на диске, который используется в качестве кэша. Вторичный кэш требует больше времени для доступа, чем первичный, находящийся в памяти. Однако для выборки объектов из файла локального диска требуется

значительно меньше времени, чем для повторного создания этих объектов при помощи исходного источника данных; поэтому предпочтительнее использовать вторичный кэш, чем не использовать его.

Вторичный кэш используют следующим образом: при заполнении основного кэша объекты не отбрасываются, а перемещаются во вторичный кэш.

Взаимодействие с программой сборки мусора

Задание постоянного ограничения количества объектов, находящихся в кэше, может являться эффективным способом обеспечения гарантии, что кэшу не потребуется дополнительный объем памяти. С другой стороны, имея возможность выбирать количество, которое, как полагают, достаточно мало, чтобы не мешать другим потребителям памяти, можно задавать размер кэша, соответствующий худшему случаю. Задавая размеры кэша, соответствующие худшему случаю, можно уменьшить производительность, которая соответствует среднему случаю. Было бы хорошо иметь способ задавать размеры кэша, соответствующие среднему случаю, а затем удалять объекты из кэша, если память нужна для других целей.

И оказалось, что такой способ есть. Он предусматривает работу с программой сборки мусора, которая может удалять объекты из кэша, если во время работы JVM обнаруживается нехватка памяти.

Программа сборки мусора имеет специальную связь с классом `java.lang.ref.SoftReference`. Ссылка на другой объект передается конструктору объекта `SoftReference`. Как только объект `SoftReference` будет создан, его метод `get` возвратит объектную ссылку, которая была передана его конструктору. Если единственная активная ссылка на объект имеется в объекте `SoftReference` и память, занимаемая объектом, нужна для других целей, то программа сборки мусора будет устанавливать ссылку в объекте `SoftReference` в null с тем, чтобы могла быть безопасно освобождена память, занимаемая объектом, на который указывает эта ссылка.

На рис. 7.33 показано участие класса `SoftReference` и связанного с ним класса `java.lang.ref.ReferenceQueue` в шаблоне Cache Management. Объект `Cache` не ссылается прямым образом на тот объект, который в нем содержится. Вместо этого он связывает каждый объект `ObjectKey` с объектом `SoftKeyReference`, первоначально содержащим ссылку на объект, идентифицируемый с помощью объекта `ObjectKey`.

Класс объекта `SoftKeyReference` является подклассом класса `java.lang.ref.SoftReference`. Если создается объект `SoftKeyReference`, он будет содержать ссылку на находящийся в кэше объект и ссылку на объект `ObjectKey`, который идентифицирует объект, содержащийся в кэше. Если в какой-то момент времени программа сборки мусора решает, что ей нужна память, которую занимает находящийся в кэше объект, и нет другой ссылки на объект, содержащийся в кэше, то программа сборки мусора очистит в объекте `SoftKeyReference` ссылку (установит ее в null) на объект, хранящийся в кэше. Программа

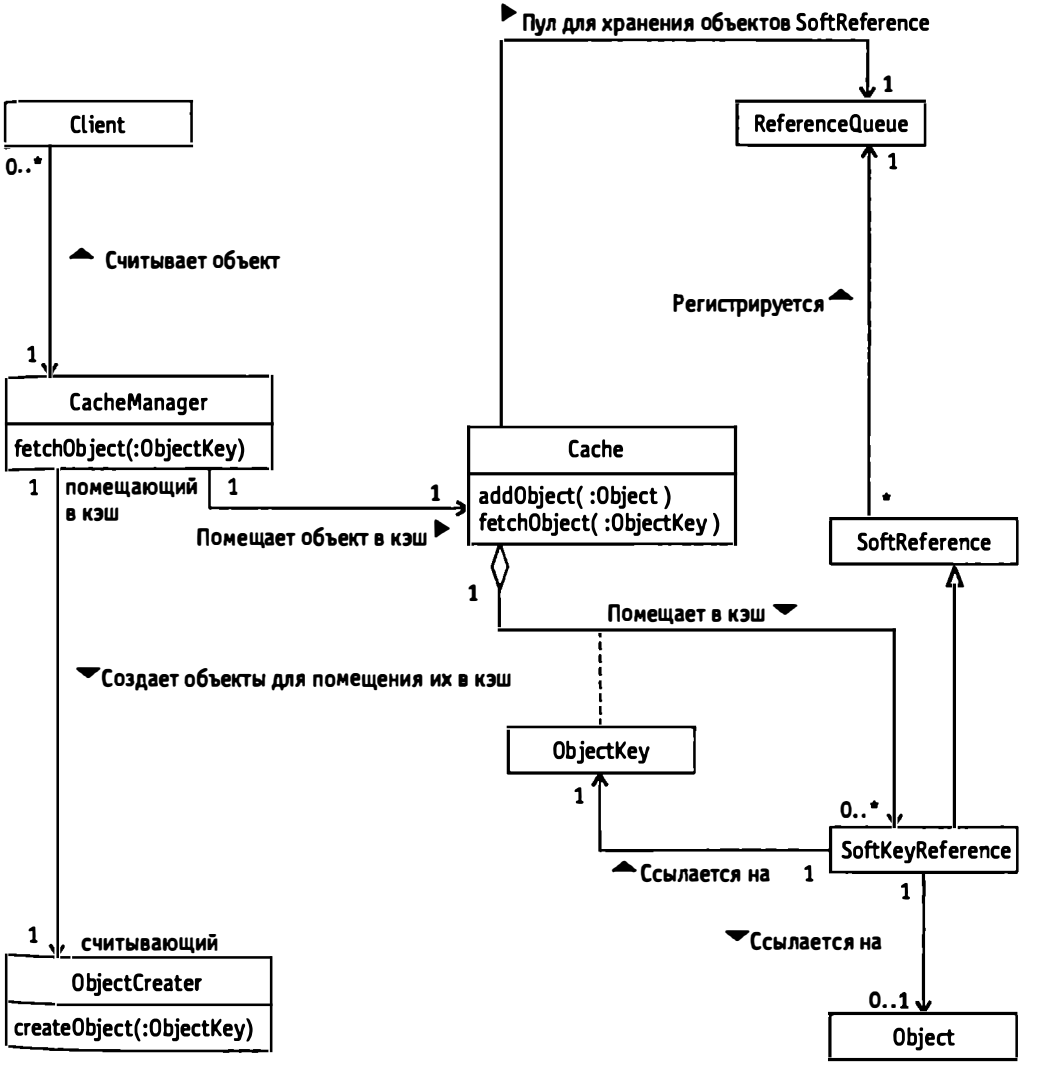


Рис. 7.33. Управление кэшем, осуществляемое с помощью класса SoftReference

сборки мусора обязательно должна выполнять очистку ссылки, перед тем как освободить память, занимаемую объектом.

Если объект `Cache` пытается получить содержащийся в кэше объект с помощью объекта `SoftKeyReference` после того, как тот был очищен, он получает null. Поскольку объект `Cache` генерирует null, если хочет сказать, что в кэше нет больше объектов, соответствующих данному объекту `ObjectKey`, очистка объекта `SoftKeyReference` свидетельствует об удалении объекта из кэша, даже если `ObjectKey` все еще находится в кэше. При помощи этого механизма объект `ObjectKey` действительно в конце концов удаляется из кэша.

Когда создается объект `SoftKeyReference`, он регистрируется при помощи объекта `ReferenceQueue` объекта `Cache`. Если программа сборки мусора очищает ссылку в объекте `SoftKeyReference`, объект `SoftKeyReference` (поскольку он регистрировался при помощи объекта `ReferenceQueue` объекта `Cache`) ставится в очередь в объекте `ReferenceQueue`. Каждый раз, когда объекту `CacheManager` поступает запрос на создание объекта, он опрашивает объект `ReferenceQueue` с целью проверки объектов `SoftKeyReference`, стоящих в очереди.

Существует некоторый объект `ObjectKey`, связанный с каждым объектом `SoftKeyReference`. Когда объект `CacheManager` получает объект `SoftKeyReference` из своего объекта `ReferenceQueue`, он удаляет объект `ObjectKey`, связанный с объектом `SoftKeyReference`, из объекта `Cache`.

СЛЕДСТВИЯ

Иногда приложения шаблона Cache Management добавляются в проект программы после того, как была обнаружена необходимость оптимизации производительности. Обычно это не представляет большой проблемы, так как влияние шаблона Cache Management на остальную часть программы минимально. Если доступ к рассматриваемым объектам уже реализован с применением шаблона Virtual Proxy, реализация шаблона Cache Management может быть помещена в класс заместителя, не требуя изменения других классов.

☺ Основным следствием применения шаблона Cache Management является то, что программа затрачивает меньше времени на получение объектов.

☺ Если объекты создаются на основе данных из внешнего источника, то другим следствием использования шаблона Cache Management является то, что данные, хранящиеся в кэше, могут стать несовместимыми с исходным источником данных. Проблема совместимости подразделяется на две отдельные проблемы, которые могут быть решены независимо друг от друга. Они получили названия *согласованности чтения* и *согласованности записи*.

Согласованность чтения означает, что в кэше всегда отражены последние изменения информации, произошедшие с исходным источником объектов. Если объекты, находящиеся в кэше, содержат информацию о курсе акций, то цены, указанные в источнике объектов, могут изменяться, и тогда цены, хранимые в кэше, больше не будут отражать текущее состояние.

Согласованность записи означает, что исходный источник объектов всегда отражает последние изменения в кэше.

Чтобы достичь абсолютной согласованности чтения или записи находящихся в кэше объектов с исходным источником объектов, необходимо реализовать механизм их синхронизации. Подобные механизмы могут реализовываться очень сложно и приводить к значительному увеличению времени выполнения. Как правило, они используют такие способы, как блокировка и оптимизационная параллельность, которые не рассматриваются в данной книге. Эти вопросы

раскрываются при описании некоторых шаблонов в книге [Grand2001]. Они упоминаются в разделе «Шаблоны проектирования, связанные с шаблоном Cache Management» в конце описания данного шаблона.

Если невозможно добиться абсолютной согласованности чтения или записи, можно задать относительную согласованность. Она не гарантирует, что содержимое кэша всегда будет совпадать с исходным источником объектов. Вместо этого она гарантирует, что при обновлении кэша или источника данных эти изменения соответствующим образом будут отражены в течение определенного промежутка времени. Более подробно этот подход рассматривается при описании шаблона Ephemeral Cache Item, которое можно найти в книге [Grand2001].

ПРИМЕР КОДА

Предположим, необходимо написать ПО для системы учета рабочего времени служащих. Система состоит из терминалов и сервера учета рабочего времени. Терминалы представляют собой небольшие устройства, вмонтированные в стену офиса. Когда служащий приходит на работу или уходит с работы, он оповещает об этом систему учета рабочего времени, проводя идентификационной карточкой через терминал учета времени. Терминал считывает идентификационный номер служащего и подтверждает правильность карточки, показывая на экране имя сотрудника и варианты выбора. Затем служащий нажимает кнопку, чтобы указать: он начинает работу, заканчивает работу, идет на перерыв или другие варианты. Терминалы передают эти события системы учета рабочего времени серверу учета рабочего времени. В конце каждого периода платежа система платежных ведомостей фирмы получает от системы учета рабочего времени количество часов работы каждого служащего и подготавливает чеки на оплату.

Все детали информации, которую работник видит на экране, зависят от профиля работника, данные которого терминал получает от сервера учета рабочего времени. Параметры служащего содержат его имя, язык, на котором выдается подсказка, и специальные опции, которые могут быть связаны с этим служащим.

Большинство фирм закрепляют за своими служащими определенное рабочее место. Такие служащие обычно используют ближайший к своему месту работы терминал учета времени. Чтобы избежать очередей перед терминалами, рекомендуется такое их расположение, чтобы один и тот же терминал учета времени использовался менее чем 70 служащими, имеющими определенное рабочее место.

Значительную часть стоимости системы учета рабочего времени составляет стоимость терминалов. Чтобы быть дешевыми, терминалы должны иметь память минимальных размеров. Это означает, что придется задать довольно умеренный максимальный размер кэша. С другой стороны, для поддержки высокой скорости реагирования желательно, чтобы терминалы сохраняли в кэше параметры служащих и почти всегда немедленно реагировали на предоставленную идентификационную карту. При установлении первоначального максимального размера кэша за основу принимается рекомендация, согласно которой

расположение терминалов должно быть таким, чтобы одним и тем же терминалом пользовалось не более 70 человек, имеющих постоянное место работы. На этом основании увеличиваем начальный размер кэша до значения, соответствующего параметрам 80 служащих.

Выбор значения, превышающего 70, объясняется тем, что в некоторых случаях одним и тем же терминалом могут воспользоваться более 70 служащих. Иногда, ввиду максимальной рабочей нагрузки, одна часть компании будет заимствовать служащих у другой части компании. Кроме того, некоторые работники, например, обслуживающий персонал, перемещаются с одного рабочего места на другое.

На рис. 7.34 представлена диаграмма классов, которая демонстрирует решение этой проблемы при помощи шаблона Cache Management.

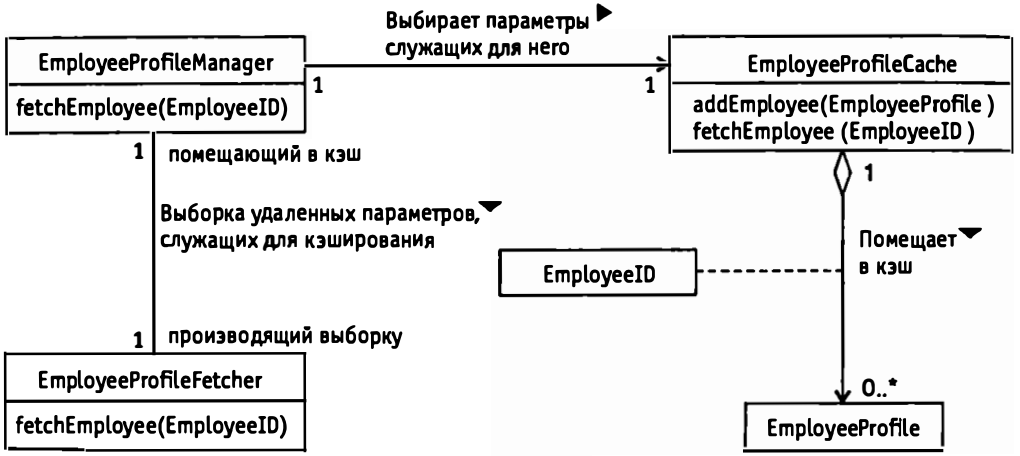


Рис. 7.34. Управление кэшем для системы учета рабочего времени

Приведем код, реализующий управление кэшем для терминала учета рабочего времени. Сначала приведем код для класса `EmployeeProfileManager`:

```
class EmployeeProfileManager {
   private EmployeeCache cache = new EmployeeCache();
   private EmployeeProfileFetcher server
      = new EmployeeProfileFetcher();

   /**
    * Из внутреннего кэша или сервера учета рабочего времени
    * (если нет во внутреннем кэше) извлекаем
    * профиль служащего, соответствующий данному
    * идентификационному номеру.
```

```
     * @return Возвратить профиль служащего или null, если
     * параметры не найдены.
     */
    EmployeeProfile fetchEmployee(EmployeeID id) {
      EmployeeProfile profile = cache.fetchEmployee(id);
      if (profile == null) {  // Если нет в кэше, пытаемся
                              // выполнить выборку из сервера.
        profile = server.fetchEmployee(id);
        if (profile != null) {  // Получаем профиль из сервера
                                // и помещаем профиль в кэш.
          cache.addEmployee(profile);
        } // if != null
      } // if == null
      return profile;
    } // fetchEmployee(EmployeeID)
  } // class EmployeeProfileManager
```

Логика, используемая классом `EmployeeProfileManager`, довольно проста. Логика класса `EmployeeCache` сложнее, так как она должна манипулировать структурой данных с целью определения, профиль какого служащего должен быть удален из кэша при добавлении профиля другого работника в заполненный кэш. Она использует также классы `SoftReference`, которые рассматривались в разделе «Реализация».

```
class EmployeeCache {
  /**
   * Для определения наименее использовавшегося
   * за последнее время профиля служащего применяется связный
   * список. Сам по себе кэш реализуется
   * при помощи объекта Hashtable. Значения Hashtable
   * представляют собой объекты связного
   * списка, которые ссылаются на реальный объект EmployeeProfile.
   */
  private Hashtable cache = new Hashtable();

  /**
   * Это головной узел связного списка, который ссылается
   * на наиболее использовавшийся в последнее время объект
   * EmployeeProfile.
   */
  LinkedList mru = null;
```

```
/**
 * Это последний узел связного списка, который ссылается
 * на наименее использовавшийся за последнее время объект
 * EmployeeProfile.
 */
LinkedList lru = null;

/**
 * Максимальное количество объектов EmployeeProfile,
 * которое может находиться в кэше.
 */
private final int MAX_CACHE_SIZE = 80;

/**
 * Количество объектов EmployeeProfile, находящихся в данный
 * момент в кэше.
 */
private int currentCacheSize = 0;

/**
 * Этот объект управляет очисткой после того, как программа
 * сборки мусора решает, что пришло время освободить память,
 * занимаемую объектом EmployeeProfile.
private CleanupQueue myCleanup = new CleanupQueue();

/**
 * Этому методу передаются объекты, предназначенные
 * для добавления в кэш. Однако этот метод на самом деле может
 * не добавить объект в кэш, если это противоречит его политике
 * добавления объектов. Этот метод может также удалять
 * находящиеся в кэше объекты, чтобы освободить место
 * для новых.
 */
public void addEmployee(EmployeeProfile emp) {
   EmployeeID id = emp.getID();
   if (cache.get(id) == null) { // Если нет в кэше.
   // Добавляем профиль в кэш,
   // делая его наиболее используемым за последнее время.
      if (currentCacheSize == 0) {
         // Рассматриваем пустой кэш как особый случай.
```

```
            lru = mru = new LinkedList(emp);
        } else {              // currentCacheSize > 0
            LinkedList newLink;
            if (currentCacheSize >= MAX_CACHE_SIZE) {
                // Удаляем наименее использовавшийся за последнее
                // время объект EmployeeProfile из кэша.
                newLink = lru;
                lru = newLink.previous;
                cache.remove(id);
                currentCacheSize-;
                lru.next = null;
                newLink.setProfile(emp);
            } else {
                newLink = new LinkedList(emp);
            } // if >= MAX_CACHE_SIZE
            newLink.next = mru;
            mru.previous = newLink;
            newLink.previous = null;
            mru = newLink;
        } // if 0
        // Помещаем профиль служащего, наиболее
        // часто использовавшийся в последнее время, в кэш.
        cache.put(id, mru);
        currentCacheSize++;
    } else {                  // Профиль уже в кэше.
        // Метод addEmployee не должен вызываться, если объект
        // уже находится в кэше. Если это случится,
        // то производится выборка,
        // чтобы объект стал наиболее используемым
        // за последнее время.
        fetchEmployee(id);
    } // if cache.get(id)
} // addEmployee(EmployeeProfile)

/**
 * Возвращает объект EmployeeProfile, связанный с данным
 * EmployeeID, или null,
 * если не найден такой EmployeeProfile.
 */
EmployeeProfile fetchEmployee(EmployeeID id) {
```

```
    // Удаляем из кэша любой EmployeeID, соответствующий
    // объект EmployeeProfile которого был удален при сборке
    // мусора.
    myCleanup.cleanup();

    LinkedList foundLink = (LinkedList)cache.get(id);
    if (foundLink == null)
      return null;            // Не в кэше.
    if (mru != foundLink) {
      if ( foundLink == lru ) {
        lru = foundLink.previous;
        lru.next = null;
      } // if lru
      if (foundLink.previous != null)
        foundLink.previous.next = foundLink.next;
      if (foundLink.next != null)
        foundLink.next.previous = foundLink.previous;
      mru.previous = foundLink;
      foundLink.previous = null;
      foundLink.next = mru;
      mru = foundLink;
    } // if currentCacheSize > 1
    return foundLink.getProfile();
  } // fetchEmployee(EmployeeID)

  /**
   * Удаляет объект EmployeeProfile, связанный с данным
   * EmployeeID, находящимся в кэше.
   */
  void removeEmployee(EmployeeID id) {
    LinkedList foundLink = (LinkedList)cache.get(id);
    if (foundLink != null) {
      if (mru == foundLink) {
          mru = foundLink.next;
      } // if mru
      if ( foundLink == lru ) {
        lru = foundLink.previous;
      } // if lru
      if (foundLink.previous != null) {
          foundLink.previous.next = foundLink.next;
```

```
      } // if foundLink.previous
      if (foundLink.next != null) {
          foundLink.next.previous = foundLink.previous;
      } // if foundLink.next
   } // if !null
} // removeEmployee(EmployeeID)

/**
 * Закрытый класс двусвязного списка для управления
 * списком наиболее использовавшихся за последнее время
 * профилей служащих. Этот класс реализует интерфейс
 * CleanupIF, поэтому его экземпляры могут быть извещены
 * после того, как программа сборки мусора
 * решит удалить объект EmployeeProfile.
 */
private class LinkedList implements CleanupIF {
   SoftReference profileReference;
   LinkedList previous;
   LinkedList next;

   LinkedList(EmployeeProfile profile) {
      setProfile(profile);
   } // constructor(EmployeeProfile)

   void setProfile(EmployeeProfile profile) {
      profileReference =
         myCleanup.createSoftReference(profile,this);
   } // setProfile(EmployeeProfile, EmployeeID)

   EmployeeProfile getProfile() {
      return (EmployeeProfile)profileReference.get();
   } // getProfile()

   /**
    * При вызове этого метода предполагается, что объект,
    * который его реализует, удаляет сам себя из любой
    * структуры данных, частью которой он является.
public void extricate() {
      EmployeeProfile profile;
      profile = (EmployeeProfile)profileReference.get();
```

```
            removeEmployee(profile.getID());
        } // extricate()
    } // class LinkedList
} // class EmployeeCache
```

Теперь приведем классы `EmployeeProfile` и `EmployeeID`:

```
class EmployeeProfile {
    private EmployeeID id; // Идентификационный номер служащего.
    private Locale locale;      // Языковые предпочтения.
    private boolean supervisor;
    private String name;        // Имя служащего.

    public EmployeeProfile(EmployeeID id,
                           Locale locale,
                           boolean supervisor,
                           String name) {
        this.id = id;
        this.locale = locale;
        this.supervisor = supervisor;
        this.name = name;
    } // Constructor(EmployeeID, Locale, boolean, String)

    public EmployeeID getID() { return id; }

    public Locale getLocale() { return locale; }

    public boolean isSupervisor() { return supervisor; }
} // class EmployeeProfile

class EmployeeID {
    private String id;

    /**
     * Constructor
     * @param id Строка, содержащая идентификационный номер
     * служащего.
     */
    public EmployeeID(String id) {
        this.id = id;
    } // constructor(String)
```

```
    /**
     * Возвращает значение хэш-кода  этого объекта.
     */
    public int hashCode() { return id.hashCode(); }

    /**
     * Возвращает true, если данный объект
     * представляет собой EmployeeID, равный этому.
     */
    public boolean equals(Object obj) {
       return ( obj instanceof EmployeeID
                && id.equals(((EmployeeID)obj).id) );
    } // equals(Object)

    /**
     * Возвращает строковое представление этого EmployeeID.
     */
    public String toString() { return id; }
} // class EmployeeID
```

А теперь — класс `org.clickblocks.dataStructure.CleanupQueue`, который используется классом `EmployeeCache` для управления высвобождением объектов `EmployeeProfile` программой сборки мусора.

```
/**
 * Этот класс инкапсулирует ReferenceQueue.
 * Он гарантирует, что объекты Reference только одного вида
 * поставлены в очередь в объект ReferenceQueue — это объекты,
 * позволяющие объектам CleanupIF, поставленным в очередь
 * в объект ReferenceQueue, удалять самих себя из любой
 * структуры данных, частью которой они являются.
 */
public class CleanupQueue {
    /**
     * ReferenceQueue, который  инкапсулирован в этом объекте.
     */
    private ReferenceQueue refQueue = new ReferenceQueue();

    /**
     * Содержит true, если не закончен вызов, активизирующий
     * очистку.
     */
    private boolean cleaning;
```

```
    /**
     * Возвращает SoftReference, предназначенный для помещения
     * в очередь ссылок, инкапсулированную в этом объекте.
     *
     * @param obj
     * Объект, на который будет указывать эта ссылка.
     * @param cleanup
     * Объект CleanupIF, метод extricate которого должен
     * вызываться после того, как SoftReference помещен в очередь.
     */
    public SoftReference createSoftReference(Object obj,
                CleanupIF cleanup) {
      return new SoftCleanupReference(obj, refQueue, cleanup);
    } // createReference(Object, CleanupIF)

    /**
     * Вызывает метод extricate всех объектов CleanupIF,
     * стоящих в очереди. Если в данный момент активизирован
     * некоторый вызов, то просто выходим из метода.
     */
    public void cleanup() {
      synchronized (this) {
        if (cleaning) {
          return;
        } // if
        cleaning = true;
      } // synchronized
      try {
        while (refQueue.poll()!=null) {
          SoftCleanupReference r;
          r = (SoftCleanupReference)refQueue.remove();
          r.extricate();
        } // while
      } catch (InterruptedException e) {
      } finally {
        cleaning = false;
      } // try
    } // cleanup()
  } // class CleanupQueue
```

И наконец, приведем интерфейс `CleanupIF`, который используется классом `CleanupQueue` для оповещения других объектов о том, что `SoftReference` был очищен.

```
public interface CleanupIF {
   /**
    * При вызове этого метода предполагаем, что объект, который
    * его реализует, удаляет сам себя из любой структуры
    * данных, частью которой он является.
    */
   public void extricate() ;
} // interface CleanupIF
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ CACHE MANAGEMENT

Facade. Шаблон Cache Management использует шаблон Facade.

Template Method. Шаблон Cache Management использует шаблон Template Method, поддерживая многократное использование своего класса `Cache` во всех областях применения.

Virtual Proxy. Шаблон Cache Management часто используется с какой-либо версией шаблона Virtual Proxy, чтобы сделать кэш прозрачным для тех объектов, которые имеют доступ к объектам в кэше.

Object Replication. Шаблон Object Replication (см. книгу [Grand2001]) затрагивает некоторые вопросы, имеющие отношение к поддержке согласованности кэша.

Optimistic Concurrency. Шаблон Optimistic Concurrency (описанный в книге [Grand2001]) представляет способ, который иногда может использоваться для управления согласованностью кэша в условиях максимальной производительности.

Ephemeral Cache Item. Шаблон Ephemeral Cache Item (представленный в книге [Grand2001]) описывает управление кэшами в условиях относительной согласованности.

ГЛАВА

Поведенческие шаблоны проектирования

Описанные в этой главе шаблоны используются для организации, управления и объединения различных вариантов поведения.

Шаблон Chain of Responsibility позволяет объекту отправлять команду, ничего не зная об объекте или объектах, получающих ее. При этом команда передается цепочке объектов, которая обычно является частью более крупной структуры. Каждый объект цепочки может обрабатывать, передавать команду следующему объекту цепочки или делать и то, и другое.

Шаблон Command инкапсулирует команды в объекте таким образом, что можно управлять их выбором и последовательностью, ставить их в очередь, отменять их и выполнять иные манипуляции.

При помощи шаблона Little Language можно искать общие, создавать сложные структуры данных и форматировать данные.

Шаблон Mediator использует один объект для согласования изменения состояний других объектов. Вместо распределения логики по разным классам он помещает логику, предназначенную для управления изменением состояний других объектов, в один класс. В результате получается более сцепленная реализация логики и более низкая связанность этих классов.

Шаблон Snapshot записывает моментальный снимок состояния объекта таким образом, чтобы это состояние можно было восстановить позднее.

Шаблон Observer позволяет объектам динамически регистрировать зависимости между объектами. В результате объект будет оповещать зависящие от него объекты об изменении своего состояния.

Шаблон State инкапсулирует состояния объекта в виде отдельных объектов, каждый из которых расширяет общий суперкласс.

Шаблон Null Object — альтернатива использованию `null` для указания отсутствия объекта, которому делегируется операция.

При помощи шаблона Strategy связанные алгоритмы инкапсулируются в классах, реализующих общий интерфейс. Это позволяет выбирать алгоритм путем определения соответствующего класса. Кроме того, этот шаблон позволяет изменять выбор алгоритма со временем.

Шаблон Template Method позволяет создать абстрактный класс, содержащий только часть логики, необходимой для выполнения задачи. Недостающая логика содержится в методах подклассов, которые замещают абстрактные методы.

Шаблон Visitor позволяет избежать усложнения классов объектов структуры, помещая всю необходимую логику в отдельный класс.

Chain of Responsibility (Цепочка ответственности)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Chain of Responsibility позволяет объекту отправлять команду, ничего не зная об объекте или объектах, получающих ее. При этом команда передается цепочке объектов, которая обычно является частью более крупной структуры. Каждый объект цепочки может обрабатывать, передавать команду следующему объекту цепочки или делать и то, и другое.

КОНТЕКСТ

Предположим, создается программа, контролирующая систему безопасности. На физическом уровне система безопасности состоит из сенсорных устройств (детекторов движения, дыма и т.д.), которые передают информацию о состоянии компьютеру. Задача компьютера состоит в том, чтобы записывать всю информацию о состояниях, отображать текущую информацию о состоянии и передавать сигналы тревоги в случае аварии.

Контролирующая программа должна обеспечивать высокую степень масштабируемости, т.е. работать в маленьких розничных магазинчиках, офисных зданиях, больших складских помещениях или в комплексах, состоящих из нескольких зданий. Это требование влияет на способ проектирования контролирующего ПО.

Чтобы не быть слишком сложной, контролирующая программа должна инстанциировать объект для каждого сенсорного устройства. Тогда способ моделирования состояния каждого сенсора будет простым. Чтобы обеспечить масштабируемость, объекты, отвечающие за отдельные сенсоры, не должны ничего знать о своем окружении, за исключением того, что они находятся на самом нижнем уровне иерархической схемы.

Схема будет содержать объекты, соответствующие таким реально существующим вещам, как комнаты, площадки, этажи и здания. Прямое моделирование реального мира — это простой способ отображения состояния различных частей здания. Кроме того, он позволяет интерпретировать состояние сенсоров, исходя из окружающей их среды. Например, если температура в закрытом помещении превышает 85°, нужно включить противопожарные разбрызгиватели только в этой комнате. Если температура на открытой площадке склада превысит 65°, то можно включить противопожарные разбрызгиватели по всей этой площади и на соседних площадках. С другой стороны, если температура

в морозильнике превышает –4°, можно включить звуковой сигнал тревоги, который даст знать, что морозильник слишком теплый.

Во всех этих случаях объект, моделирующий сенсор, сам не решает, что делать с состоянием сенсора. Вместо этого он делегирует принятие решения объекту, находящемуся на более высоком уровне иерархии и обладающему большей информацией о контексте. Такие объекты либо решают, что делать с извещением, либо передают его объекту, стоящему на более высоком уровне иерархии.

На рис. 8.1 показан пример иерархической организации объектов.

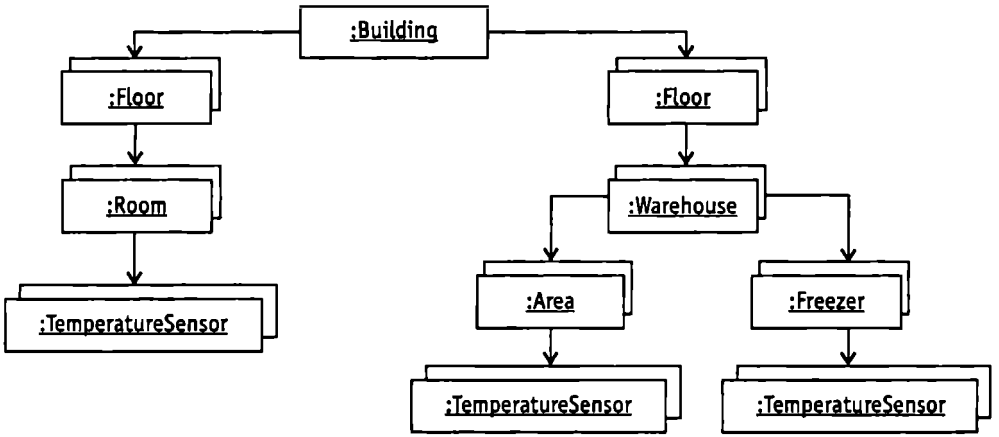


Рис. 8.1. Объекты физической безопасности

Например, если объект `TemperatureSensor`, находящийся на территории склада, получает от физического сенсора извещение о текущей температуре, то он передает это извещение содержащему его объекту `Area`. Вместо принятия решения по поводу критичности температуры `Area` передает извещение содержащему его объекту `Warehouse`. Объект `Warehouse` оценивает температуру. Если она превышает 65°, объект `Warehouse` принимает решение, что возник пожар. Он включает разбрызгиватели в той области, которая его оповестила, и в близлежащих областях. Объект `Warehouse` не передает извещения о температуре.

МОТИВЫ

☺ Нужно, чтобы объект мог посылать команду другому объекту, не идентифицируя получателя. Объекта-отправителя беспокоит не то, какой объект обработает команду, а только то, что некоторый объект должен получить команду и обработать ее.

☺ Необходимо, чтобы получатели команды имели возможность обрабатывать команду, ничего не зная об объекте, который ее послал.

☺ Получать и обрабатывать команду могут несколько объектов, поэтому нужен способ задания приоритетов получателей, не загружая объект-отправитель сведениями об этих получателях.

☺ Объекты, способные обрабатывать команды, организованы в виде структуры, которая может использоваться для задания приоритетов потенциальных обработчиков команд.

РЕШЕНИЕ

На рис. 8.2 представлена диаграмма классов, описывающая организацию шаблона Chain of Responsibility. Опишем роли, исполняемые классами и интерфейсом в этом шаблоне.

CommandSender. Экземпляры класса, выступающего в роли `CommandSender`, отправляют команды первому объекту цепочки объектов, способных обработать команду. Он посылает команду, вызывая метод `postCommand` первого объекта `CommandHandlerIF`.

CommandHandlerIF. Все объекты цепочки, имеющие возможность обрабатывать команду, должны реализовывать интерфейс, выступающий в этой роли. Он определяет два метода:

1. Метод `handleCommand` для обработки любых команд, которые должны обрабатываться реализующим классом. Метод `handleCommand` возвращает true, если он обработал команду, или false, если не обработал.
2. Метод `postCommand`, который вызывает метод `handleCommand`. Если метод `handleCommand` возвращает false и в цепочке имеется следующий объект, он вызывает метод `postCommand` этого объекта. Если метод `handleCommand` возвращает true, это значит, что не нужно передавать команду следующему объекту цепочки.

AbstractCommandHandler. Классы, играющие эту роль, являются абстрактными классами, реализующими метод `postCommand`. Это позволяет воспользоваться преимуществами общей реализации метода `postCommand` в классах, выступающих в роли `ConcreteCommandHandler`. Обычно классы используют логику по умолчанию для метода `postCommand`. Классы, выступающие в роли `CommandSender`, должны ссылаться на объекты цепочки ответственности не как на экземпляры класса `AbstractCommandHandler`, а только через интерфейс `CommandHandlerIF`. Класс `AbstractCommandHandler` — это просто деталь реализации. Хотя это и не обычная практика, классы могут реализовывать интерфейс `CommandHandlerIF` и не являться подклассами класса `AbstractCommandHandler`.

ConcreteCommandHandler1, ConcreteCommandHandler2 и т.д. Экземпляры классов в этой роли представляют собой объекты цепочки ответственности, которые могут обрабатывать команды.

Как правило, объекты `CommandHandler` являются частью более крупной структуры (рис. 8.2).

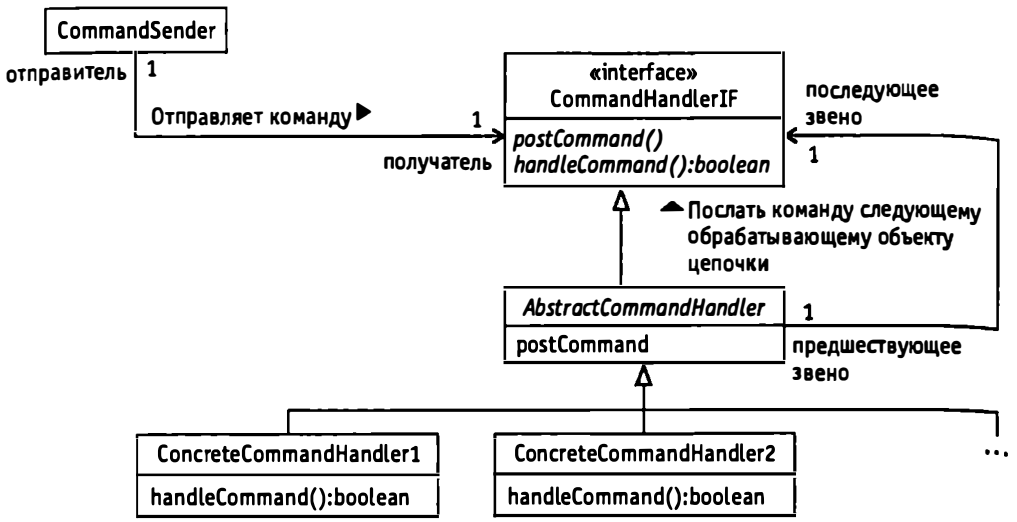


Рис. 8.2. Шаблон Chain of Responsibility

РЕАЛИЗАЦИЯ

Во многих случаях объекты, составляюшие цепочку ответственности, являются частью более крупной структуры, и цепочка ответственности образуется при помощи некоторых звеньев этой более крупной структуры. Если звенья, необходимые для образования цепочки ответственности, еше не сушествуют, нужно добавить в классы переменные экземпляра и методы доступа с целью создания связей, образуюших цепочку ответственности.

Всякий раз при реализации шаблона Chain of Responsibility необходимо решать, как передавать команды самой цепочке объектов и как по ней. Сушествуют два основных способа решения этой задачи. Один заключается в том, чтобы инкапсулировать команды всех видов в единственном объекте, который может быть передан единственному методу `postCommand`. Другой — в том, чтобы иметь столько различных типов методов `postCommand` и `handleCommand`, сколько существует разных видов информации, связанной с командами.

Лучшим вариантом обычно является передача команд в единственном объекте. Он берет на себя все затраты по созданию объектов, сводя к минимуму затраты на передачу параметров методам следующего объекта цепочки. При этом максимально снижаются затраты на прохождение команды по цепочке объектов. Передача команд в единственном объекте, как правило, требует меньшего кода.

С другой стороны, передача информации, составляюшей команду, при помощи отдельных параметров избавляет от затрат на создание объекта за счет затрат на передачу дополнительных параметров. Если известно, что цепочка объектов будет короткой, оптимальным вариантом является передача команды в виде набора параметров.

СЛЕДСТВИЯ

☺ Шаблон Chain of Responsibility уменьшает связанность между объектом, отправляющим команду, и объектом, обрабатывающим команду. Отправитель команды не должен знать о том, какой объект реально обработает команду. Он просто должен иметь возможность отправить команду объекту, находящемуся в начале цепочки ответственности.

☺ Шаблон Chain of Responsibility допускает гибкость во время принятия решения, каким образом обрабатывать команды. Решение о том, какой объект будет обрабатывать команду, может меняться при изменении объектов, входящих в цепочку ответственности, или при изменении последовательности объектов в цепочке ответственности.

• Шаблон Chain of Responsibility не гарантирует обработку каждой команды. Необработанные команды игнорируются.

☹ Если количество объектов в цепочке слишком велико, то могут быть проблемы с эффективностью, касающиеся времени, необходимого для прохождения команды по цепочке. Высокий процент необработанных команд усугубляет проблему, так как необработанные команды проходят по всей длине цепочки.

Модель делегирования событий

Модель делегирования событий предусматривает использование объектов трех видов:

- источники событий представляют собой объекты, которые являются источниками информации о возникновении события некоторого вида;
- приемники событий представляют собой объекты, которые должны знать о появлении события некоторого определенного вида;
- события — это объекты, которые инкапсулируют информацию о появлении события.

Объекты-источники событий передают объекты-события объектам-приемникам событий. Объекты событий являются экземплярами подкласса класса `java.util.EventObject`.

Существует соглашение об именах, которое задает структуру модели делегирования событий. Проиллюстрируем это соглашение при помощи событий действий.

События `Action` представляются классом `ActionEvent`. Чтобы экземпляры некоторого класса могли получать события действий, этот класс должен реализовывать интерфейс `ActionListener`. Такие интерфейсы,

как `ActionListener`, объявляют один или несколько методов, не имеющих возвращаемого значения, которым будет передан соответствующий вид события. Классы, которые являются источниками событий действий, определяют методы `addActionListener` и `removeActionListener`. Эти методы позволяют объектам регистрироваться и отменять регистрацию на получение событий действий от объекта, который является источником событий действий.

Более подробная информация о модели делегирования событий содержится в спецификации JavaBeans, которую можно найти на сайте http://java.sun.com/products/javabeans/docs/spec.html.

ПРИМЕНЕНИЕ В JAVA API

В версии 1.0 языка Java использовался шаблон Chain of Responsibility для обработки событий пользовательского интерфейса. При этом в качестве цепочки ответственности применялась иерархия контейнеров пользовательского интерфейса. Если событие отправлялось кнопке или другому компоненту GUI, он должен был либо обработать событие, либо передать его своему контейнеру. Хотя эта схема была работоспособной, существовало достаточно проблем, поэтому создатели языка Java предприняли решительные шаги по изменению модели событий в языке Java. Две самые серьезные проблемы были связаны с эффективностью и гибкостью.

1. Некоторые платформы генерируют множество событий, которые большинство GUI или не обрабатывают, или не проявляют к ним никакого интереса, например, событие `MOUSE_MOVE`. Оно может генерироваться каждый раз, когда мышь перемещается хотя бы на один пиксель. Некоторые программы, созданные с использованием первоначальной модели событий, заметно замедлялись при каждом быстром перемещении мыши, так как они тратили много времени на передачу событий `MOVE_MOVE` (которые никогда не обрабатывались) через иерархию контейнеров.

2. Шаблон Chain of Responsibility предполагает, что все объекты, обрабатывающие команду, являются экземплярами общего суперкласса или реализуют общий интерфейс. Тогда программа должна отправлять команды только экземплярам этого общего суперкласса или интерфейса. Первоначальная модель событий в языке Java требовала от каждого объекта, способного обрабатывать событие, быть экземпляром общего суперкласса `Component`. Это означало, что невозможно было прямо направлять события объектам, не являющимся объектами GUI, поскольку только объекты GUI — экземпляры класса `Component`.

Вторая проблема на самом деле обернулась преимуществом для некоторых приложений. Шаблон Chain of Responsibility делает невыгодной передачу команд объектам, которые не являются частью цепочки обработчиков. Для не-

которых приложений передача команды объекту вне цепочки обработчиков почти всегда вызывает ошибку. Пример системы физической безопасности, рассмотренный в разделе «Контекст», представляет собой пример такого приложения. Для приложений такого рода использование шаблона Chain of Responsibility позволяет делать меньше ошибок, чем применение модели делегирования события.

Еще одно преимущество шаблона Chain of Responsibility по сравнению с моделью делегирования событий состоит в том, что он позволяет явным образом контролировать порядок передачи событий обработчикам.

ПРИМЕР КОДА

На рис. 8.3 представлены классы, используемые в примере системы физической безопасности.

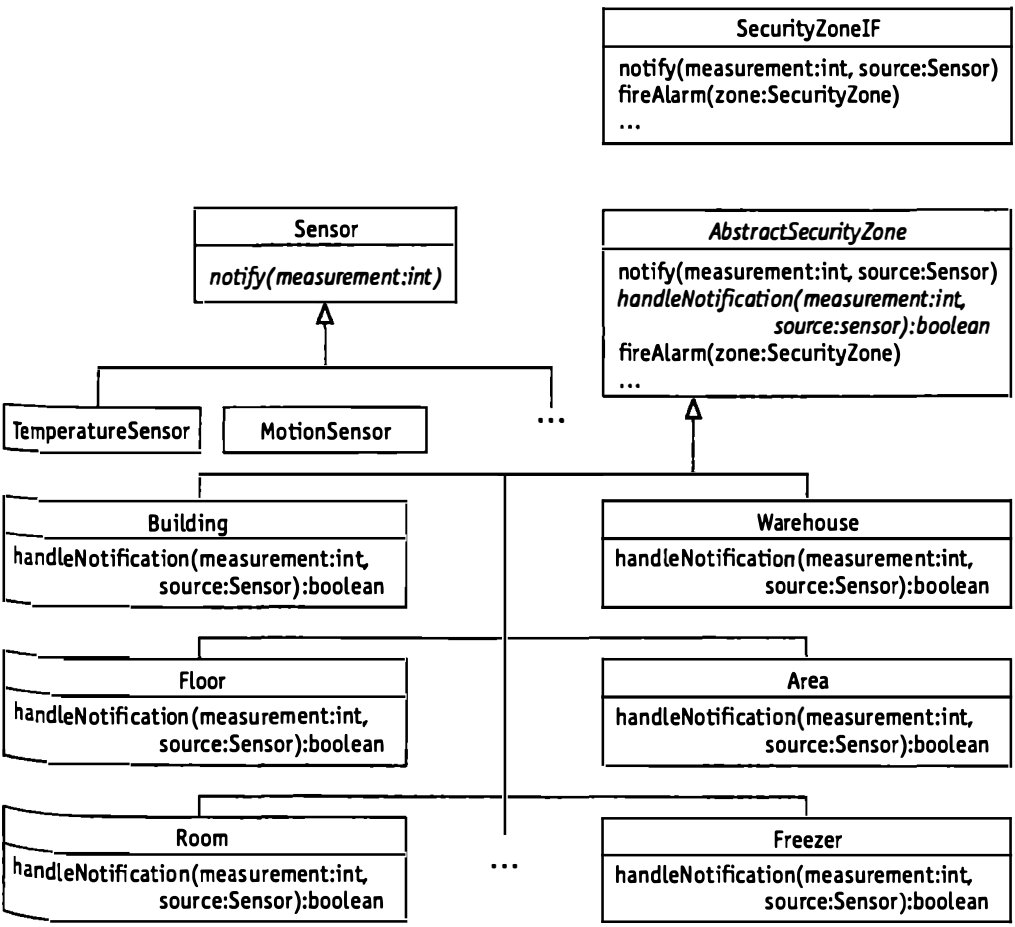


Рис. 8.3. Классы системы физической безопасности

Классы, которые расширяют класс Sensor, вызывают метод notify. Этот метод они наследуют от него с целью передачи измеренного значения объекту, который отвечает за обработку измерений. Классы, которые расширяют класс AbstractSecurityZone, отвечают за обработку измеренных значений, полученных от соответствующего объекта Sensor.

Приведем пример кода для классов, показанных на рис. 8.3. Первым представлен код для класса TemperatureSensor. Обратите внимание, что класс TemperatureSensor ничего не делает с данными, полученными от температурного сенсора, а передает их дальше.

```
class TemperatureSensor extends Sensor {
   private SecurityZone zone;
...
   /**
    * Если датчик температуры, связанный с объектом,
    * фиксирует изменение температуры, вызывается метод notify.
    */
   void notify(int measurement) {
      zone.notify(measurement, this);
   } // notify(int)
} // class TemperatureSensor
```

Все классы, которые моделируют зоны безопасности, реализуют интерфейс SecurityZoneIF:

```
public interface SecurityZoneIF {
   /**
    * Этот метод вызывается для оповещения зоны контроля
    * безопасности об изменении значения сенсора.
    */
   public void notify(int measurement, Sensor source) ;

   /**
    * Этот метод вызывается дочерней зоной для сообщения
    * о пожаре.
    */
   public void fireAlarm(SecurityZone zone) ;
} // interface SecurityZoneIF
```

Следующий код — для класса SecurityZone, который является суперклассом для всех классов, образующих цепочки ответственности в этом примере:

```
abstract class SecurityZone implements SecurityZoneIF {
   private SecurityZone parent;
```

```
  /**
   * Возвращает родительскую зону этого объекта.
   */
  SecurityZone getParent() {
    return parent;
  } // getParent()

  /**
   * Вызываем этот метод для оповещения зоны
   * о новом измеренном значении сенсора.
   */
  public void notify(int measurement. Sensor sensor) {
    if (!handleNotification(measurement, sensor)
        && parent != null) {
      parent.notify(measurement, sensor);
    } // if
  } // notify(int, Sensor)

  /**
   * Этот метод вызывается методом notify, поэтому
   * объект может обработать измерения.
   */
  protected
  abstract boolean handleNotification(int measurement,
                                      Sensor sensor);

  /**
   * Этот метод вызывается дочерней зоной для сообщения
   * о пожаре.
   * Предполагается, что зона потомка должна включить
   * разбрызгиватели или предпринять другие меры по контролю
   * над огнем в пределах своей зоны. Цель этого метода состоит
   * в том, чтобы он был замещен в подклассах и мог
   * предпринимать любые необходимые меры за пределами зоны
   * потомка.
   */
  public void fireAlarm(SecurityZone zone) {
    // Включаем разбрызгиватели.
...
    if (parent != null)
      parent.fireAlarm(zone);
  } // fireAlarm(SecurityZone)
} // class SecurityZone
```

Теперь коды подклассов класса `SecurityZone`, рассмотренных в разделе «Контекст»:

```
class Area extends SecurityZone {
...
   /**
    * Этот метод вызывается методом notify,
    * поэтому объект может обработать измерения.
    */
   boolean handleNotification(int measurement,Sensor sensor){
      if (sensor instanceof TemperatureSensor) {
         if (measurement > 150) {
            fireAlarm(this);
            return true;
         } // if
      } // if
...
      return false;
   } // handleNotification(int, Sensor)
} // class Area

class Warehouse extends SecurityZone {
   ...
   /**
    * Этот метод вызывается методом notify,
    * поэтому объект может обработать измерения.
    */
   protected
   boolean handleNotification(int measurement,Sensor sensor){
...
      return false;
   } // handleNotification(int, Sensor)

   public void fireAlarm(SecurityZone zone) {
      if (zone instanceof Area) {
         // Включаем разбрызгиватели в близлежащих областях.
         ...
         // Не вызываем super.fireAlarm, так как при этом
         // включаются разбрызгиватели по всему складу.
         if (getParent() != null)
            getParent().fireAlarm(zone);
         return;
```

```
        } // if
...
        super.fireAlarm(zone);
    } // fireAlarm(SecurityZone)
} // class Warehouse
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ CHAIN OF RESPONSIBILITY

Composite. Если цепочка объектов, используемая шаблоном Chain of Responsibility, является частью более крупной структуры, то эта структура создается при помощи шаблона Composite.

Command. Шаблон Chain of Responsibility не идентифицирует объект, выполняющий команду, а шаблон Command явно и конкретно определяет такой объект.

Template Method. Если объекты, образующие цепочку ответственности, являются частью более крупной структуры, созданной при помощи шаблона Composite, то шаблон Template Method может использоваться для планирования поведения отдельных объектов.

Command (Команда)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Command инкапсулирует команды в объекте таким образом, что можно управлять их выбором и последовательностью, ставить их в очередь, отменять их и выполнять иные манипуляции.

КОНТЕКСТ

Предположим, нужно спроектировать программу обработки текста таким образом, чтобы она могла выполнять какие-то действия или отменять их. С этой целью реализуется действие в виде объекта, имеющего методы `do` и `undo` (рис. 8.4).

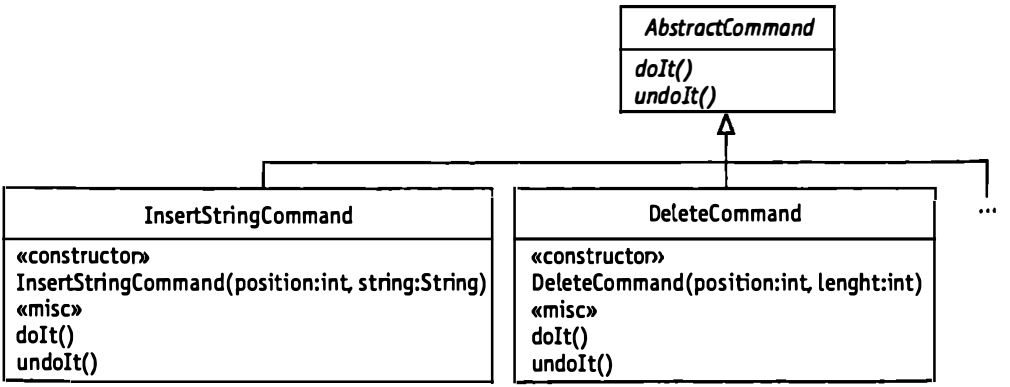


Рис. 8.4. Классы do и undo

Если дать текстовому процессору задание, вместо непосредственного выполнения команды он создаст экземпляр подкласса класса `AbstractCommand`, соответствующий команде. Текстовый процессор передает всю необходимую информацию конструктору этого экземпляра. Например, если поступает команда вставить один или более символов, он создает объект `InsertStringCommand` и передает конструктору объекта ту позицию в документе, где нужно сделать вставку, и строку, которую нужно вставить.

Если процессор обработки текста однажды «материализовал» команду в виде объекта, для выполнения этой команды он вызывает метод `doIt` этого объекта.

Кроме того, текстовый процессор помещает объект команды в структуру данных, которая позволяет текстовому процессору сохранять историю выполненных команд и, в связи с этим, отменять команды в порядке, обратном их вызову. С этой целью вызываются их методы `undo`.

МОТИВЫ

- ☺ Нужно управлять последовательностью, выбором и согласованностью выполнения команд.
- ☺ Необходимо управлять отменой и повторным выполнением команд.
- ☺ Нужно поддерживать непрерывную регистрацию выполненных команд. Можно создать для этого журнал посредством расширения объектов команд таким образом, чтобы их методы `doIt` и `undoIt` создавали регистрационные записи. Можно использовать базы данных при ведении журнала для отмены результатов ранее выполненных команд, и поэтому такой журнал может быть встроен в механизм управления транзакциями, чтобы разрешить отмену команд при отмене транзакции.

РЕШЕНИЕ

На рис. 8.5 представлена диаграмма классов, участвующих в шаблоне Command.

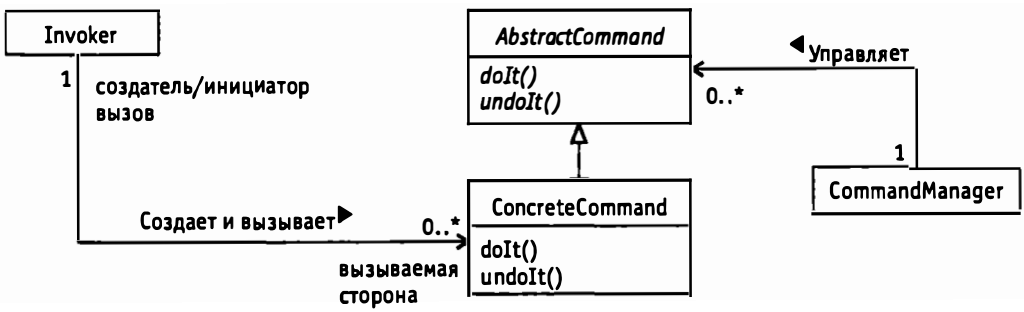


Рис. 8.5. Шаблон Command

Опишем роли, исполняемые этими классами.

AbstractCommand. Класс в этой роли представляет собой суперкласс для классов, инкапсулируюших команды. Он обычно определяет абстрактный метод `doIt`, вызываемый другими классами для выполнения команды, инкапсулированной в подклассах класса `AbstractCommand`. Если нужна поддержка команды отмены, класс `AbstractCommand` определяет также метод `undoIt`, который отменяет результаты последнего вызова метода `doIt`.

ConcreteCommand. Классы в этой роли представляют собой конкретные классы, инкапсулирующие определенную команду. Другие классы вызывают команду,

обращаясь к методу `doIt` класса. Обращение к логике отмены команды осуществляется посредством вызова метода `undoIt` класса.

Конструктор объекта обычно предоставляет любые параметры, необходимые для команды. Большинство команд требуют, по крайней мере, одного параметра — объекта, на который воздействует команда. Например, команда сохранения объекта на диске обычно требует, чтобы сохраняемый объект передавался конструктору объекта команды.

Invoker. Если нужно вызвать команду, класс, исполняющий эту роль, создает конкретные объекты команд. Он может вызвать метод `doIt` этих объектов или оставить эту работу для объекта `CommandManager`.

CommandManager. Класс `CommandManager` отвечает за управление коллекцией объектов команд, созданной объектом `Invoker`. В круг конкретных обязанностей класса `CommandManager` может входить управление командами отмены и повтора, упорядочение и планирование команд.

Классы `CommandManager` обычно не зависят от тех приложений, которые их применяют, и могут использоваться очень часто.

РЕАЛИЗАЦИЯ

При реализации шаблона Command необходимо рассмотреть несколько вопросов. Первый и, возможно, самый важный заключается в том, чтобы решить, какими будут команды. Если команды выдаются пользовательским интерфейсом, предоставляющим команды пользовательского уровня, то вполне естественный способ идентификации классов конкретных команд состоит в том, чтобы иметь конкретный класс команды для каждой команды пользовательского уровня. Если придерживаться этой стратегии и при этом имеются некоторые особенно сложные пользовательские команды, то сложность классов команд будет соответствующей. Чтобы избежать излишнего усложнения одного класса, нужно реализовывать более сложные команды пользовательского уровня при помощи нескольких классов команд.

Если количество внешних команд или команд пользовательского уровня слишком велико, то можно использовать стратегию, согласно которой они реализуются при помощи комбинаций объектов команд. Эта стратегия позволяет реализовать большое количество внешних команд при помощи меньшего количества классов команд.

Undo/redo

При реализации необходимо рассмотреть еще один вопрос: получение информации о состоянии, необходимой для команд отмены (undo). Чтобы отменять результаты действия команды, нужно сохранять достаточную часть состояния объектов, на которые она воздействует; тогда возможно восстановление этого состояния.

Бывают команды, отменить которые невозможно, так как для этого нужно сохранять огромное количество информации о состоянии. Например, команда глобального поиска и замены иногда может изменять такое большое количество информации, что поддержка всей первоначальной информации потребует недопустимо большого объема памяти. Некоторые команды вообще нельзя отменять из-за невозможности восстановления состояния, измененного этими командами, например, команды удаления файлов.

Объект `CommandManager` должен знать, когда выполняемая команда не может быть отменена. Это объясняется рядом причин.

- Предположим, что объект `CommandManager` отвечает за первоначальное выполнение команд. Если перед выполнением команды этот объект знает, что она не может быть отменена, то, используя обычный механизм, он может предупреждать пользователя о том, что подлежашая выполнению команда не должна отменяться. Предупреждая пользователя, он может также предложить ему не выполнять команду.
- Сохранение истории команд с целью их отмены требует памяти, а иногда и других ресурсов. После выполнения команды, которая не может быть отменена, история команд может быть удалена. Сохранение истории команд после выполнения неотменяемой команды требует ненужного расхода ресурсов.
- Большинство пользовательских интерфейсов программ, имеющих команду отмены, содержат в меню пункт, который пользователь может выбрать для отмены команды. Для пользователя будет неприятным сюрпризом, если он попытается отменить действие, а в ответ получит сообщение, что последняя команда не может быть отменена. Чтобы не раздражать пользователя, объект управления командой должен разрешать или запрещать в меню пункт отмены, если последняя выполненная команда, соответственно, может или не может быть отменена.

Если не нужно поддерживать операции отмены, можно упростить шаблон. В таком случае класс `AbstractCommand` не должен определять метод `undoIt`.

Как избежать зависимости от пользовательских интерфейсов

Существует широко распространенное расширение шаблона Command, применяемое в том случае, когда команды выдаются пользовательским интерфейсом. Цель этого расширения шаблона заключается в том, чтобы избежать привязки компонентов пользовательского интерфейса к конкретному объекту команды или даже не позволить компонентам пользовательского интерфейса знать о каких-либо конкретных классах команд. Расширение предусматривает встраивание имени команды в компоненты пользовательского интерфейса и использование шаблона Factory Method для создания объектов команд (рис. 8.6).

В таком случае классы GUI-компонентов ссылаются на имя команды, которую они вызывают, а не на класс команды, реализующий эту команду, и не на

экземпляр этого класса. Они вызывают команды косвенно, передавая имя команды объекту фабрики, который создает экземпляр соответствующего класса конкретной команды.

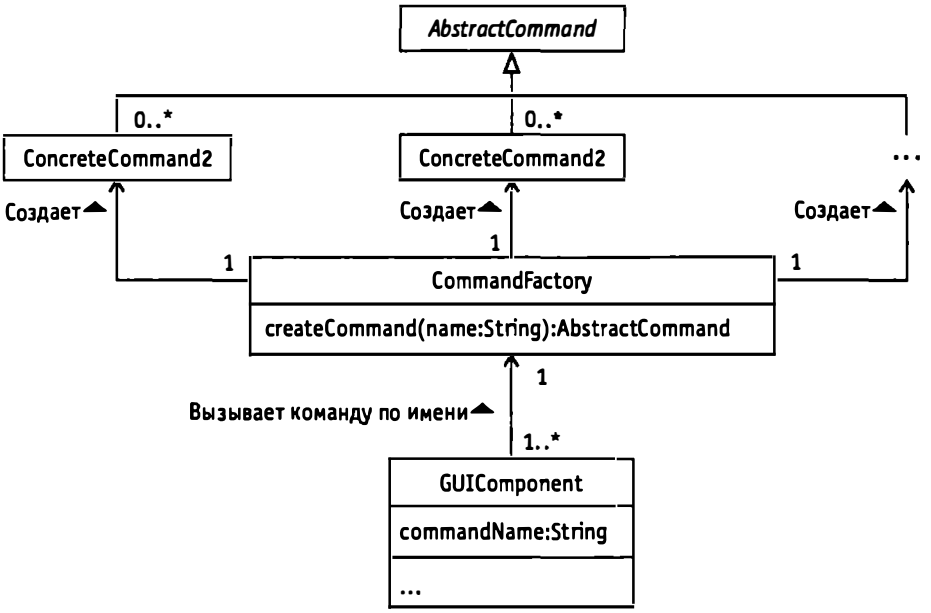


Рис. 8.6. Создание объектов команд при помощи Factory Method

Вызов команд через фабрику команд приводит к появлению некоторой косвенности, которая может быть очень полезна. Косвенность позволяет нескольким объектам, выдающим команду, совместно использовать один и тот же объект команды. Еще более важно то, что косвенность упрощает использование настраиваемых пользователем меню и панелей инструментов.

Как правило, программы создают команды в ответ на события, например, нажатие клавиш или выбор пункта меню. Некоторые программы используют события для представления команд. Эту идею нельзя назвать слишком удачной, поскольку изменения пользовательского интерфейса могут предполагать изменение команд, вызываемых пользовательским интерфейсом.

СЛЕДСТВИЯ

☺ Вызов команды и ее выполнение производятся разными объектами. Такое разделение обеспечивает гибкость, необходимую для согласования и упорядочения команд. Материализация команд в виде объектов означает, что они, подобно любым другим объектам, могут быть представлены в виде коллекции, им можно делегировать операции и выполнять разные манипуляции.

☺ Возможность создания коллекций и управления последовательностью команд означает, что шаблон Command может быть положен в основу механизма, поддерживающего макросы клавиатуры. Такой механизм позволяет записывать последовательность команд и позднее воспроизводить ее. Кроме того, шаблон Command может использоваться для создания сложных шаблонов других видов.

☺ Добавление новых команд обычно не вызывает затруднений, так как не нарушает каких-либо зависимостей.

ПРИМЕНЕНИЕ В JAVA API

В Java API нет каких-либо удачных примеров шаблона Command. Но оно содержит некоторое подобие шаблона Command в JFC (Java Foundation Classes, библиотека базовых классов Java). Классы кнопок или пунктов меню имеют методы `getActionCommand` и `setActionCommand`, которые можно использовать для получения и задания имени команды, связанной с кнопкой или пунктом меню.

ПРИМЕР КОДА

Продолжим рассмотрение примера команд отмены и повтора для текстового процессора, представленного в разделе «Контекст».

На рис. 8.7 изображена диаграмма стандартного взаимодействия, в ходе которого создаются и выполняются команды.

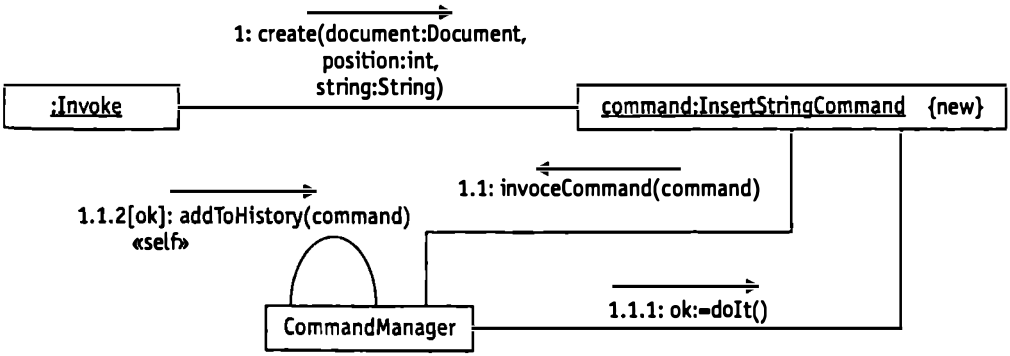


Рис. 8.7. Взаимодействие в шаблоне Command для текстового процессора

Объект создает экземпляр класса `InsertStringCommand`, передавая его конструктору документ, в который нужно вставить строку, саму строку и позицию, куда ее надо вставить. После инициализации объекта `InsertStringCommand` конструктор вызывает метод `invokeCommand` объекта `CommandManager`. Метод

invokeCommand вызывает метод doIt объекта InsertStringCommand, который реально вставляет строку. Если метод doIt возвращает true, указывая, что команда прошла успешно и может быть отменена, то объект CommandManager добавляет объект InsertStringCommand в свою историю команд.

Приведем код, реализующий вышесказанное. Сначала — исходный код класса AbstractCommand, который является суперклассом для всех классов команд в данном примере:

```
public abstract class AbstractCommand {
   public final static CommandManager manager
   = new CommandManager();

   /**
    * Выполняет команду, инкапсулированную в этом объекте.
    * @return Возвращает true при успешном выполнении
    * и возможности отмены.
    */
   public abstract boolean doIt();

   /**
    * Отменяет последний вызов метода doIt.
    * @return Возвращает true, если отмена прошла успешно.
    */
   public abstract boolean undoIt();
} // class AbstractCommand
```

Класс AbstractCommand создает экземпляр класса CommandManager, используемый для управления всеми экземплярами класса AbstractCommand. Конкретные подклассы класса AbstractCommand могут получать доступ к объекту CommandManager через переменную manager класса AbstractCommand.

Приведем исходный код для конкретных подклассов класса AbstractCommand:

```
class InsertStringCommand extends AbstractCommand {
...
   /**
    * Constructor
    */
   InsertStringCommand(Document document,
                       int position,
                       String strng) {
     this.document = document;
     this.position = position;
     this.strng = strng;
     manager.invokeCommand(this);
   } // Constructor(Document, int, String)
```

```
    /**
     * Выполняет команду, инкапсулированную в этом объекте.
     * @return Возвращает true, если этот вызов doCommand был
     * успешным и его можно отменять.
     */
    public boolean doIt() {
       try {
          document.insertStringCommand(position, strng);
       } catch (Exception e) {
          return false;
       } // try
       return true;
    } // doIt()

    /**
     * Отменяет команду, инкапсулированную в этом объекте.
     * @return Возвращает true, если отмена была успешной.
     */
    public boolean undoIt() {
       try {
          document.deleteCommand(position, strng.length());
       } catch (Exception e) {
          return false;
       } // try
       return true;
    } // undoIt()
} // class InsertStringCommand
```

Большинство других подклассов класса `AbstractCommand` имеют такую же базовую структуру. Исключениями являются классы для команд отмены и повтора. Они будут представлены в данном разделе ниже.

Теперь приведем код для класса `CommandManager`, который отвечает за управление выполнением команд. Кроме того, для текстового процессора экземпляры этого класса обязаны поддерживать историю команд для отмен и повторов. Обратите внимание на специальное управление командами отмены и повтора.

```
class CommandManager {
   // Максимальное количество команд, сохраняющихся в истории.
   private int maxHistoryLength = 100;
   private LinkedList history = new LinkedList();
   private LinkedList redoList = new LinkedList();
```

```
/**
 * Вызывает команду и добавляет ее в историю.
 */
public void invokeCommand(AbstractCommand command) {
   if (command instanceof Undo) {
      undo();
      return;
   } // if undo
   if (command instanceof Redo) {
      redo();
      return;
   } // if redo
   if (command.doIt()) {
      // Метод doIt возвращает true, что означает возможность
      // отмены команды.
      addToHistory(command);
   } else { // Команда не может быть отменена,
            // поэтому история команд очищается.
      history.clear();
   } // if
   // После выполнения команды (кроме команд отмены или
   // повтора) убеждаемся, что список команд повтора пуст.
   if (redoList.size() > 0)
      redoList.clear();
} // invokeCommand(AbstractCommand)

private void undo() {
   if (history.size() > 0) { // Если история не пуста.
       AbstractCommand undoCmd;
       undoCmd = (AbstractCommand)history.removeFirst();
       undoCmd.undoIt();
       redoList.addFirst(undoCmd) ;
   } // if
} // undo()

private void redo() {
   if (redoList.size() > 0) { // Если список повтора не пустой.
       AbstractCommand redoCommand;
       redoCmd = (AbstractCommand)redoList.removeFirst();
       redoCmd.doIt();
       history.addFirst(redoCmd) ;
   } // if
} // redo()
```

```
    /**
     * Добавляет команду в историю команд.
     */
    private void addToHistory(AbstractCommand command) {
      history.addFirst(command);
      // Если размер истории превысил значение maxHistoryLength,
      // удаляет из истории самую старую команду.
      if (history.size() > maxHistoxyLength)
      history.removeLast();
    } // addToHistory(AbstractCommand)
} // class CommandManager
```

Обратите внимание, что класс `CommandManager` в действительности не использует классы команд, представляющие классы отмены и повтора. Он просто проверяет, реализуют ли их экземпляры интерфейсы `Undo` или `Redo`. Эти интерфейсы представляют собой чистые маркеры-интерфейсы. Приведем листинг интерфейса `Undo`:

```
interface Undo {
} // interface Undo
```

Исходный код для интерфейса `Redo` выглядит так же.

Эти маркеры-интерфейсы предназначены для того, чтобы поддерживать класс `CommandManager` независимым от конкретных подклассов класса `AbstractCommand`. Поскольку класс `CommandManager` обязан управлять отменой и повтором, все эти классы, представляющие отмену и повтор, дают объекту `CommandManager` знать, что он должен отменить или повторить последнюю команду. Лучше всего, если он будет делать это так, чтобы не требовать от класса `CommandManager` какой-либо специальной логики. Тогда должен использоваться механизм, который позволит объекту `CommandManager` спрашивать, не нужно ли выполнить отмену или повтор, а не получать извещение об этом. Если можно определить, реализует ли некоторый объект интерфейс `Undo` или `Redo`, то объект `CommandManager` может спрашивать, нужно выполнить отмену или повтор, ничего не зная о классе, реализующем этот интерфейс.

И наконец, приведем исходный код для класса `UndoCommand`. Класс `RedoCommand` отличается от него незначительно.

```
class UndoCommand extends AbstractCommand implements Undo {
    /**
     * Эта реализация метода doIt на самом деле ничего не делает.
     * Логика для отмены находится в классе CommandManager.
     */
    public boolean doIt() {
```

```
        // Этот метод не должен вызываться никогда.
        throw new NoSuchMethodError();
    } // doIt()

    /**
     * Эта реализация метода undoIt на самом деле ничего не
     * делает.
     * Команды undo не отменяют. Вместо этого
     * выдается команда redo.
     */
    public boolean undoIt() {
        // Этот метод не должен вызываться никогда.
        throw new NoSuchMethodError();
    } // undoIt()
} // class UndoCommand
```

Поскольку не должно быть обращений к методам данного класса, эти методы всегда генерируют исключение.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ COMMAND

Factory Method. Шаблон Factory Method иногда используется для установления некоторой косвенности между пользовательским интерфейсом и классами команд.

Little Language. Шаблон Command можно использовать как вспомогательное средство при реализации шаблона Little Language. Кроме того, шаблон Little Language может использоваться для реализации команд.

Marker Interface. Шаблон Marker Interface может использоваться вместе с шаблоном Command для реализации обработки команд отмены и повтора.

Snapshot. Если нужно задать упрощенный механизм отмены, согласно которому состояние объекта сохраняется полностью, а не команда за командой, то можно использовать шаблон Snapshot.

Template Method. Шаблон Template Method может использоваться для реализации логики отмены высшего уровня в рамках шаблона Command.

Little Language (Малый язык)

В основе шаблона Little Language лежит шаблон Interpreter, описанный в работе [GoF95], и понятие малого языка, которое было популяризировано Джоном Бентли в работе [Bentley86]. Более сложные способы проектирования и реализации языков можно найти в работе [ASU86].

СИНОПСИС

Предположим, необходимо решить несколько похожих проблем и их решения могут быть выражены в виде различных комбинаций немногочисленных элементов или операций. Самый простой способ описания решения таких проблем состоит в том, чтобы определить малый язык. Поиск общих, создание сложных структур данных и форматирование данных — это наиболее часто встречающиеся проблемы, которые решаются при помощи малого языка.

КОНТЕКСТ

Допустим, нужно написать программу, которая выполняет поиск в коллекции файлов с целью обнаружения файлов, содержащих заданную комбинацию символов или слов. Чтобы не писать отдельную программу для каждой операции поиска, можно определить малый язык, при помощи которого пользователи опишут комбинации слов, а затем написать одну программу, которая будет находить файлы с комбинациями слов, заданными при помощи малого языка.

Определение языка обычно состоит из двух частей. *Синтаксис* языка задает символы и слова, составляющие язык, и способы их комбинирования. *Семантика* языка задает значение символов, слов и их комбинаций, входящих в состав языка.

Синтаксис языка обычно определяется грамматикой. Грамматика — это набор правил, определяющих последовательность символов, образующих слова и условные обозначения языка. Кроме того, грамматика содержит правила, которые позволяют комбинировать символы и слова языка с целью создания более крупных конструкций.

Точное определение семантики большого языка может быть очень сложным и длинным. Однако для малого языка иногда вполне достаточно нескольких простых поясняющих абзацев.

Малый язык можно определить следующим образом. Сначала создается несколько примеров того, как язык должен выглядеть. Затем примеры обобщаются с целью получения окончательного определения.

Следуя этому плану, рассмотрим некоторые моменты, которые полезно учесть в малом языке при задании комбинаций слов. Основное — это возможность задавать комбинацию, которая состоит из одного слова. Самый очевидный способ задания для поиска комбинации такого рода заключается в том, чтобы просто написать это слово, например, так:

```
bottle
```

Кроме того, может появиться необходимость задавать комбинации слов, которые не содержат определенное слово. Это делается просто: перед словом, которого не должно быть в комбинациях слов, нужно поставить слово «not»:

```
not box
```

Тогда задаются для поиска все комбинации слов, которые не содержат слова «box».

Слова, подобные слову «not», которые задают определенное условие, называются *зарезервированными*, поскольку они предназначены для специальных целей и не могут использоваться таким же образом, как обычные слова. Если слово рассматривается как зарезервированное, нельзя задать комбинацию слов для поиска, содержащую такое слово, простым написанием этого слова. Однако существует способ указания комбинации слов, которая содержит любое произвольное слово, последовательность слов или знаки пунктуации. Для этого такую комбинацию слов заключают в кавычки:

```
"Yes, not the"
```

Рассмотрим задание комбинации из двух слов. Очевидно, что синтаксис для такой комбинации должен позволять задавать сами слова, входящие в комбинацию. Поскольку комбинировать слова можно по-разному, синтаксис должен предоставить также способы комбинации слов. Для этого необходимо написать одно слово комбинации, далее указать специальное слово, которое показывает, как комбинируются слова, а вслед за ним — второе слово комбинации. Например, чтобы указать комбинации слов, содержащих хотя бы одно из слов «bottle» или «jar», можно написать следующее:

```
bottle or jar
```

Для задания других способов сочетания двух слов понадобятся дополнительные слова:

- «and» для указания комбинаций слов, содержащих оба слова;
- «near» для указания комбинаций слов, в которых между указанными словами могут находиться до 25 каких-то других слов.

Если нужно, например, использовать зарезервированное слово «and» вместе с зарезервированным словом «not» для указания комбинации слов, содержащей «garlic» и не содержащей «onions», необходимо будет написать

```
garlic and not onions
```

Эти примеры охватывают почти все, что нужно для описания комбинаций, содержащих два слова. При использовании более двух слов потребуется рассмотрение дополнительных условий. Очевидно, что

```
red and "pickup truck" and broken
```

означает комбинации слов, содержащих все три элемента: слово «red», фразу «pickup truck» и слово «broken».

Если смешиваются разные способы сочетания слов, цель становится неоднозначной. Словосочетание

```
turkey or chicken and soup
```

указывает на комбинацию слов, содержащую слово «turkey» или оба слова «chicken» и «soup»? Или оно описывает комбинацию слов, содержащую слово «soup» и хотя бы одно из слов «chicken» или «turkey»? Чтобы преодолеть эту неоднозначность, нужно поставить скобки, определяющие порядок использования логических связей в комбинации слов. Для задания первого варианта интерпретации можно написать:

```
turkey or (chicken and soup)
```

Для указания второго варианта можно написать:

```
(turkey or chicken) and soup
```

Большинству пользователей не нравится, когда их заставляют писать скобки, поэтому желательно найти способ решения проблемы без использования скобок. Для устранения неоднозначности такого рода в определениях языка широко применяется *правило предшествования*, которое назначает приоритеты различным используемым в языке операциям. Его суть заключается в том, чтобы предоставить способ определения порядка операций. Операции с более высоким приоритетом выполняются перед операциями с более низким приоритетом. Допустим, назначают следующие значения приоритетов:

near	3
and	2
or	1

При таком определении значений приоритетов словосочетание

```
manison or big near house and rich
```

задает комбинации слов, содержащие слово «manison» или оба слова «rich» и «big» при условии, что расстояние между словом «big» и словом «house» не превышает 25 слов.

Если бы использовались скобки, то такое условие выглядело бы следующим образом:

```
manison or ((big near house) and rich)
```

Перед тем как спроектировать классы и придать смысл малому языку, важно описать грамматику, которая определит синтаксис языка. На основе такого

описания можно начинать кодирование. Грамматика может быть организована в соответствии с различными стратегиями. Во всех примерах данного шаблона используется *нисходящая стратегия*. Это значит, что начинают с языковой конструкции самого верхнего уровня — комбинации слов — и затем определяют все конструкции более низких уровней, которые могут входить в ее состав, до тех пор пока грамматический разбор не будет завершен полностью.

Выше уровня отдельных символов находится уровень конструкций, образующих грамматику и называющихся *лексемами*. Лексемы бывают либо терминальными, либо нетерминальными. Терминальные лексемы соответствуют непрерывной последовательности символов, а также строкам в кавычках и скобках. Например, слово в виде

```
fence
```

представляет собой терминальную лексему. Конструкции более высокого уровня, состоящие из терминальных лексем, называются *нетерминальными лексемами*. Комбинация слов — это нетерминальная лексема.

В большинстве малых языков, включая данный язык словосочетаний, терминальные лексемы могут отделяться друг от друга символами пробелов, которые не несут смысловой нагрузки.

Правила, позволяющие распознавать последовательности символов как терминальные лексемы, называются *правилами лексического анализа*. Правила, позволяющие распознавать нетерминальные лексемы в виде последовательностей терминальных и нетерминальных лексем, называются *продукциями*.

Система обозначений, используемая в данной книге для записи продукций, называется *формой Бэкуса-Наура*. Согласно этой форме терминальные и нетерминальные лексемы записываются разными шрифтами, что позволяет различать их. В этой книге терминальные лексемы указываются так:

```
quoted_string
```

а нетерминальные:

```
combination
```

Продукция, содержащая нетерминальную лексему и последовательность терминальных и нетерминальных лексем, может быть распознана как первая нетерминальная лексема. Приведем пример продукции:

```
combination → word
```

Эта продукция говорит о том, что нетерминальная лексема комбинации может содержать просто терминальную лексему слова.

Если существует несколько последовательностей лексем, которые могут быть распознаны как нетерминальная лексема, то будет существовать несколько продукций для такой нетерминальной лексемы. Для каждой последовательности, которая может быть распознана как такая нетерминальная лексема, будет существовать одна продукция. Например, следующий набор продукций задает синтаксис для комбинаций, которые содержат или не содержат конкретное слово.

combination → **word**

combination → **not word**

Способ, который используется для определения того, что нетерминальная лексема должна распознаваться из последовательности лексем неопределенной длины, называется *рекурсией*. Приведем описание набора продукций, который охватывает почти весь синтаксис предыдущих примеров:

combination → (*combination*)

combination → *simpleCombination*

combination → *simpleCombination* **or** *combination*

combination → *simpleCombination* **and** *combination*

combination → *simpleCombination* **near** *combination*

simpleCombination → **word**

simpleCombination → **not word**

Обратите внимание, что четыре продукции из пяти для *combination* являются рекурсивными. Три продукции из этих четырех могли быть записаны при помощи *combination* в качестве первой нетерминальной лексемы и при помощи *simpleCombination* — в качестве второй нетерминальной лексемы. В любом случае они должны соответствовать одним и тем же последовательностям лексем. Однако для способа реализации, описанного ниже в данном разделе при включении продукций в код, разница существует. Для того способа, который будет рассматриваться, доказано, что при написании продукций лучше всего использовать правую рекурсию. *Правая рекурсия* означает, что, если есть возможность выбора, в каком месте продукции применить рекурсию, мы должны выбрать такой вариант, когда рекурсия используется как можно правее.

Хотя этот набор продукций не описывает всех деталей данного языка словосочетаний, содержащейся в нем информации достаточно для того, чтобы можно было работать, используя пример, приведенный в книге. Теперь рассмотрим, каким образом использовать такие продукции для распознавания следующей строки как комбинации

```
fox and not brown
```

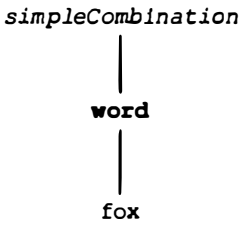
Взглянув на продукции для *combination*, можно увидеть, что *combination* может начинаться либо с открывающей скобки, либо с *simpleCombination*. Строка начинается с лексемы *word*. Поскольку строка, которую мы пытаемся распознать как *combination*, не начинается с открывающей скобки, мы пытаемся распознать начало строки как *simpleCombination*. Это соответствует продукции

simpleCombination → **word**

У нас распозналась следующая часть строки:

```
fox and not brown
```

Линия над строкой показывает, какая часть строки была идентифицирована. Вот что у нас распозналось:



Иными словами, распозналась лексема *simpleCombination*, которая состоит из лексемы слова, содержащей слово *fox*. Нам нужно распознать именно *combination*. Четыре продукции для *combination* начинаются с *simpleCombination*. Одна из этих продукций —

combination → *simpleCombination*

Теперь у нас есть возможность выбора: сравнивать эту продукции с тем, что уже распозналось, или попытаться подобрать более длинную продукцию для *combination*. В таком случае всегда лучше подобрать более длинную продукцию. Если нельзя подобрать более длинную продукцию, то возвращаемся и подбираем более короткую.

Следующая лексема в строке — **and**. Существует только одна продукция для *combination*, начинающаяся с *simpleCombination*, за которой следует **and**.

combination → *simpleCombination* **and** *combination*

Чтобы завершить сопоставление строки с этой продукцией, нам необходимо распознать остальную часть строки как *combination*.

Следующая лексема в строке — **not**. Взглянув на продукции для *combination*, можно увидеть, что *combination* может начинаться либо с открывающей скобки, либо с *simpleCombination*. Поскольку строка, которую мы пытаемся распознать как *combination*, не начинается с открывающей скобки, пробуем распознать начало строки как *simpleCombination*. Для *simpleCombination* есть продукция, которая начинается с лексемы **not**. Теперь попытаемся завершить подбор продукции

simpleCombination → **not word**

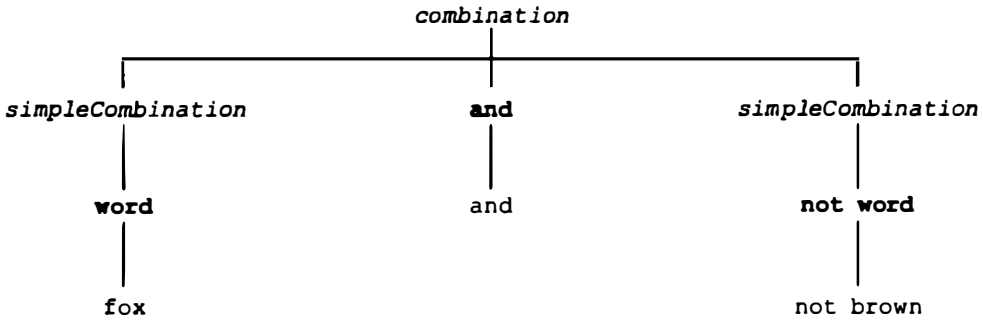
и заканчиваем сопоставление

combination → *simpleCombination* **and** *combination*

Поскольку мы идентифицировали половину продукции *simpleCombination*, предполагаем, что следующей лексемой в строке должна быть лексема *word*. У нас распозналась следующая часть строки:

`fox and not brown`

Следующая лексема — **word**. Значит, мы успешно подобрали продукции, которые пытались сопоставить. Кроме того, на этом содержимое строки исчерпано, то есть мы распознали всю строку как комбинацию, имеющую следующую внутреннюю структуру:



Такая структура в виде дерева, полученная в процессе синтаксического разбора строки, называется *деревом синтаксического разбора*. Для большинства языков реализация производится проще и быстрее, если сначала строится структура данных в виде дерева синтаксического разбора, а затем это дерево синтаксического разбора используется для определения следующих действий.

Набор продукций, применявщийся в предыдущем примере, не учитывает все особенности языка словосочетаний. Он не позволяет включать в комбинацию строки в кавычках. Кроме того, он не определяет правила предшествования для **and**, **near** или **or**.

Рассмотренный набор продукций использует одну и ту же нетерминальную лексему для сопоставления последовательностей лексем **word** и **not word**. Это значит, что после построения дерева синтаксического разбора объект одного и того же типа будет представлять оба вида этих последовательностей. Если можно будет определить, какой тип последовательности представляет объект, взглянув только на тип этого объекта, то дерево синтаксического разбора будет интерпретироваться проще и быстрее. Тогда продукции будут распознавать такие последовательности как две разные нетерминальные последовательности.

Опишем полный набор продукций, учитывающий изложенные выше уточнения:

combination → *orCombination*

orCombination → *andCombination* **or** *orCombination*

orCombination → *andCombination*

andCombination → *nearCombination* **and** *andCombination*

andCombination → *nearCombination*

nearCombination → *simpleCombination* **near** *nearCombination*

nearCombination → *simpleCombination*

simpleCombination → **(** *orCombination* **)**

simpleCombination → *wordCombination*

simpleCombination → *notWordCombination*

wordCombination → **word**

wordCombination → **quoted_string**

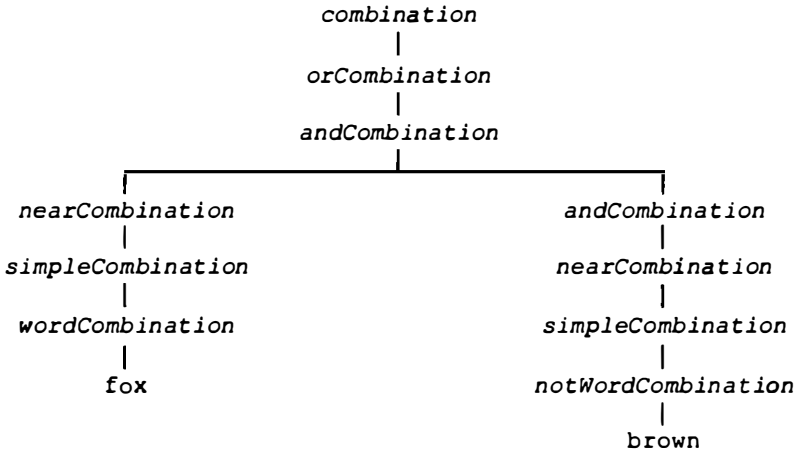
notWordCombination → **not word**

notWordCombination → **not quoted_string**

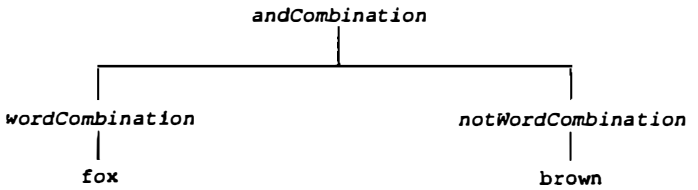
Если использовать эти продукции для синтаксического разбора строки, создавая объект узла дерева синтаксического разбора для каждой нетерминальной лексемы, то потребуется больше объектов узлов дерева синтаксического разбора, чем при использовании предыдущего набора продукций. Опишем дерево синтаксического разбора, которое было бы построено при синтаксическом разборе той же строки, которая рассматривалась в предыдущем примере

```
fox and not brown
```

но теперь используем последние продукции при создании объекта узла дерева синтаксического разбора для каждой нетерминальной лексемы:



Обратите внимание, что последнее дерево синтаксического разбора содержит много узлов, которые не имеют никакой полезной информации. Это дерево без потери информации можно упростить так:



Перед написанием любого кода нужно решить, какие продукции являются вспомогательными, а какие несут в себе полезную информацию. Синтаксиче-

ский анализатор должен создавать узлы дерева синтаксического разбора, содержащие информацию.

Мы рассмотрели основы написания продукций для распознавания лексем. Теперь опишем способы задания правил лексического анализа, определяющих, как распознавать терминальные лексемы в последовательности символов.

Во многих малых языках лексические правила бывают настолько простыми, что можно адекватно определять их посредством набора описаний, выполненных на естественном языке, например:

- интервал содержит один или более последовательно расположенных знаков пробелов, табуляций, перехода на новую строку или конца строки. Интервал может использоваться для разделения терминальных лексем. Он не имеет другого значения и отбрасывается;
- лексема **and** состоит из следующей последовательности трех букв: **a**, **n**, **d**;
- ...

Можно, конечно, отдать предпочтение более точному методу, основанному на регулярных выражениях. Регулярные выражения представляют собой способ, который определяет порядок совпадения последовательности символов. Например, регулярное выражение

```
[0-9]+
```

соответствует последовательности, состоящей из одной или более цифр. Можно использовать разнообразные системы обозначений регулярных выражений, но они должны быть достаточно выразительными для большинства малых языков. В табл. 8.1 приведено описание системы обозначений регулярных выражений, применяющейся в данном параграфе для определения лексических правил языка словосочетаний.

Таблица 8.1. Язык словосочетаний

Регулярное выражение	С чем оно совпадает
c	Символом *c*, если *c* не является одним из специальных символов, описанных ниже
\	Если за \ следует одна из управляющих последовательностей, которую язык Java разрешает использовать в строках, то это означает то же самое, что и данная управляющая последовательность. Если за \ указаны какие-либо символы, которые считаются специальными в регулярных выражениях, то в паре символов второй символ не считается специальным. Например, \\ совпадает с символом обратной наклонной черты
.	Любым символом
^	Началом последовательности или строки
$	Концом последовательности или строки

Таблица 8.1. Окончание

Регулярное выражение	С чем оно совпадает
[*s*]	Символом, находящимся в наборе и диапазонах символов. Например, `[aeiouy]` совпадает с гласными нижнего регистра. `[A-Za-z_]` совпадает со всеми буквами верхнего или нижнего регистра или с подчеркиванием
[^*s*]	Символом, не находящимся в наборе символов и в диапазонах символов. Например, `[^0-9]` совпадает с символом, который не является цифрой. Такая трактовка символа ^ соответствует случаю, когда символ стоит справа от [. Например, [+^] совпадает со знаком плюс или циркумфлексом
*r**	Нулем или более случаями появлениями регулярного выражения *r*
r+	Одним или более случаями появления регулярного выражения *r*
r?	Нулем или одним случаем появления регулярного выражения *r*
rx	Совпадает, если совпадает регулярное выражение *r* и следующее за ним регулярное выражение *x*
(*r*)	Совпадает, когда совпадает регулярное выражение *r*. Например, `(xyz)*` совпадает с нулевым или более количеством появлений последовательности *xyz*
r|*x*	Любой строкой, которая содержит регулярное выражение *r* или регулярное выражение *x*. Например, `(abc)|(xyz)` совпадает с последовательностью *abc* или последовательностью *xyz*
r{*m*,*n*}	По крайней мере, с *m*, но не более чем с *n* количеством появлений регулярного выражения *r*

В табл. 8.2 представлен набор правил лексического анализа для языка словосочетаний. В первом столбце находится регулярное выражение. Второй столбец содержит имя терминальной лексемы или любое выражение, которое считается опознанным, если оно совпадает с регулярным выражением. Если синтаксический анализатор должен обнаружить у себя на входе следующую терминальную лексему, он будет проверять регулярные выражения в порядке их появления до тех пор, пока не обнаружит одно, совпадающее с входными данными. Если ни одно регулярное выражение не совпало с входными данными, то синтаксический анализатор знает, что входные данные неверны.

Если регулярное выражение совпадает с некоторыми входными данными и существует терминальная лексема, заданная во втором столбце, то синтаксический анализатор распознает эту лексему и обрабатывает ее в соответствии с продукцией, которую он пытается подобрать. Если во втором столбце нет терминальной лексемы, то входные данные, совпадающие с регулярным выражением, отбрасываются, и синтаксический анализатор опять начинает с первого регулярного выражения.

Таблица 8.2. Правила лексического анализа для языка словосочетаний

Регулярное выражение	Имя терминальной лексемы или опознанное выражение
`[\u0000-\u0020]+`	пробел, т.е. игнорируется
`[Oo][Rr]`	**or**
`[Aa][Nn][Dd]`	**and**
`[Nn][Ee][Aa][Rr]`	**near**
`[Nn][Oo][Tt]`	**not**
`[a-zA-Z0-9]+`	**word**
`\(`	**(**
`\)`	**)**
`"([^"]*(\\")*)*"`	**quoted_string**

Мы определили синтаксис языка словосочетаний и рассмотрели семантику языка. Теперь нам нужно спроектировать классы. На рис. 8.8 представлены классы, необходимые для реализации языка словосочетаний.

Экземпляры класса `LexicalAnalyzer` читают символы комбинации слов из экземпляра класса `InputStream`. Экземпляр класса `Parser` читает лексемы из экземпляра класса `LexicalAnalyzer`, вызывая его метод `nextToken`. Как показано на диаграмме, метод `nextToken` возвращает объект `TerminalToken`. Однако на диаграмме нет класса `TerminalToken`. Если бы он был, то являлся бы абстрактным классом, не определяющим ни методов, ни переменных. Каждый из подклассов класса `TerminalToken` соответствовал бы отдельной терминальной лексеме. Подклассы класса `TerminalToken` не должны были бы задавать ни методов, ни переменных (или только одну переменную, содержащую строку, которая была бы опознана как эта терминальная лексема). Такие объекты были бы упрощенными, инкапсулирующими терминальную лексему только одного типа, и в некоторых случаях — строку. Реализации шаблона `Little Language` обычно не берут на себя труд инкапсулировать такого рода информацию. Реализации класса `LexicalAnalyzer` обычно предоставляют эту часть информации в виде неинкапсулированных фрагментов информации.

По мере того как объект `Parser` получает лексемы от объекта `LexicalAnalyzer`, он создает экземпляры подклассов класса `Combination`, выстраивая их в виде дерева синтаксического разбора.

Класс `Combination` представляет собой абстрактный суперкласс для всех классов, которые инстанциируются для создания узлов дерева синтаксического разбора. Класс `Combination` определяет абстрактный метод `contains`, который принимает в качестве аргумента строку и возвращает массив чисел типа int.

Подклассы класса `Combination` замещают метод `contains` с целью определения, удовлетворяются ли требования некоторого подкласса, предписывающие содержание нужной комбинации слов. Если строка содержит нужную комбинацию слов, метод `contains` возвращает массив целочисленных значений, которые представляют собой смещения слов в этой строке. Если строка не содержит нужной комбинации слов, то метод `contains` возвращает `null`.

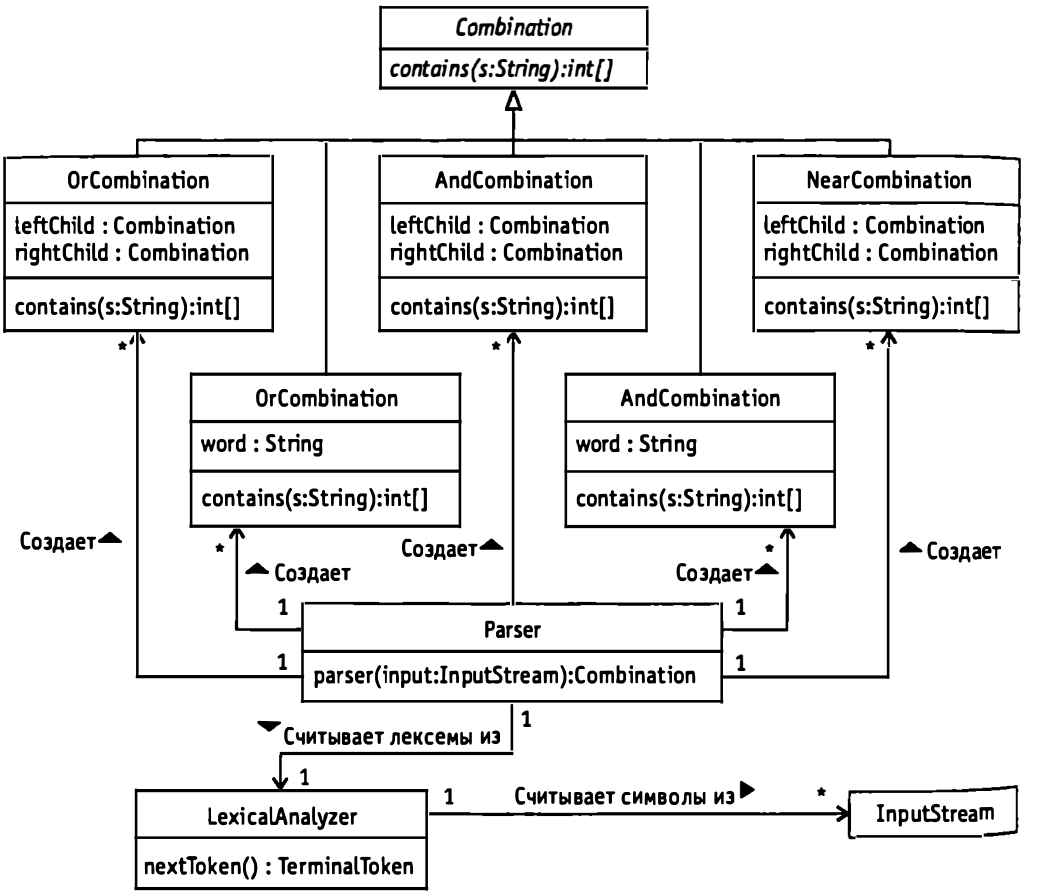


Рис. 8.8. Классы языка словосочетаний

МОТИВЫ

☺ Нужно идентифицировать, создавать или форматировать данные похожих типов, используя множество различных комбинаций немногочисленных операций.

☺ Простое представление комбинаций операций может обеспечить приемлемую производительность.

РЕШЕНИЕ

Шаблон Little Language начинается с проекта малого языка, который может задавать комбинации операций, необходимых для решения специальных проблем. Проект малого языка определяет его синтаксис при помощи продукций и лексических правил, описание которых приводится в разделе «Контекст». Семантика малого языка обычно задается неформальным образом и описывает назначение конструкций малого языка на английском или другом естественном языке.

Следующая стадия после определения малого языка — проектирование классов, используемых для реализации языка. На рис. 8.9 представлена организация классов, принимающих участие в шаблоне Little Language.

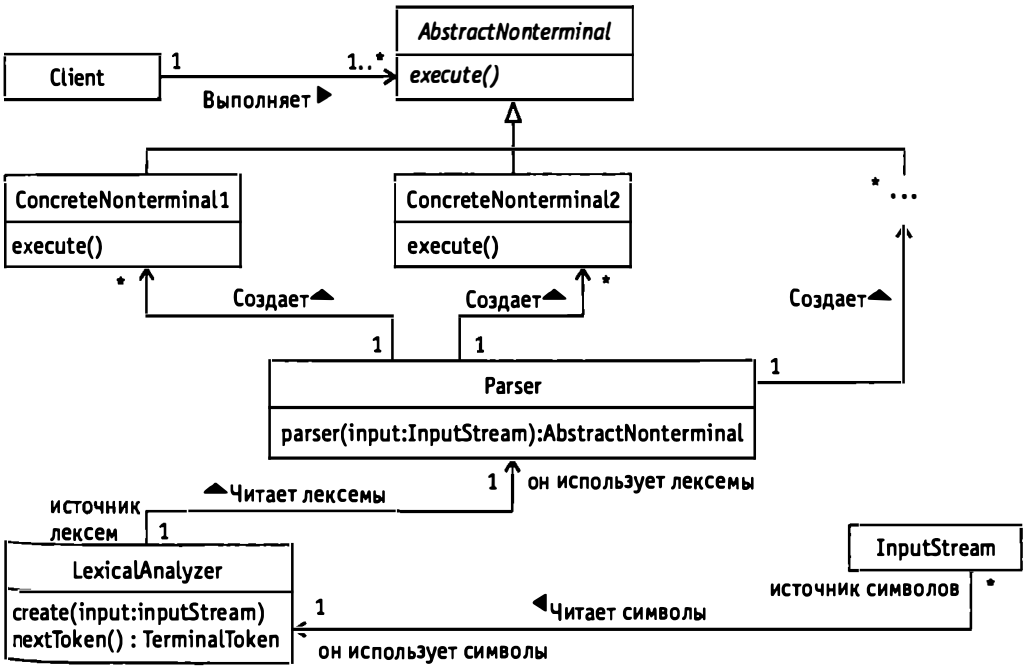


Рис. 8.9. Классы шаблона Little Language

Опишем роли, исполняемые этими классами.

Client. Экземпляр класса в этой роли запускает программу малого языка, предоставляет ей все необходимые данные и использует результаты, которые получены в результате работы программы. Он создает экземпляр класса `Parser` для программ синтаксического разбора, которые он предоставляет при помощи объектов `InputStream`. Метод `parse` объекта `Parser` возвращает экземпляр класса `AbstractNonterminal` объекту `Client`. Этот объект является корневым в дереве синтаксического разбора. Объект `Client` вызывает метод `execute` объекта `AbstractNonterminal` для запуска программы.

Lexical Analyzer. При вызове метод `parse` объекта `Parser` создает объект `LexicalAnalyzer` для чтения символов из того объекта `InputStream`, который был ему передан. Объект `LexicalAnalyzer` считывает символы из объекта `InputStream`, распознает при помощи лексических правил найденные им терминальные лексемы и возвращает эти лексемы классу `Parser` посредством вызова метода `nextToken` объекта `LexicalAnalyzer`. Метод `nextToken` возвращает следующую терминальную лексему, обнаруженную во входных данных.

Parser. Объект `Client` создает объект класса `Parser` и затем вызывает метод `parse` объекта `Parser` для выполнения синтаксического разбора входных данных, полученных от объектов `InputStream`, сравнивая лексемы входных данных с грамматическими продукциями. По мере того как он подбирает продукции, метод `parse` строит дерево синтаксического разбора и возвращает ссылку на корень дерева синтаксического разбора объекту `Client`.

AbstractNonterminal. Класс в этой роли является абстрактным суперклассом для всех классов, экземпляры которых могут представлять узлы дерева синтаксического разбора.

Объект `Client` вызывает абстрактный метод `execute` этого класса для выполнения программы.

ConcreteNonterminal1, ConcreteNonterminal2 и т.д. Экземпляры классов, исполняющих эти роли, представляют собой узлы дерева синтаксического разбора.

TerminalToken. Этот абстрактный класс не задает ни методов, ни переменных. Его подклассы представляют терминальные лексемы, распознаваемые классом `LexicalAnalyzer`.

InputStream. Экземпляр класса `java.io.InputStream` может использоваться для чтения потока символов.

РЕАЛИЗАЦИЯ

Некоторые программисты полагают, что при реализации синтаксического анализатора лучше распределить большую часть его логики по множеству классов, соответствующих нетерминальным лексемам или даже продукциям. Объясняют это программисты чаще всего тем, что такая структура является почему-то более объектно-ориентированной. Однако подобную структуру синтаксических анализаторов сложнее поддерживать, чем структуру, описанную выше. Это объясняется следующими двумя причинами.

1. Распределение логики синтаксического разбора по нескольким классам приводит в результате к меньшему сцеплению классов, что затрудняет их понимание. Синтаксический анализатор для малых языков обычно достаточно мал, поэтому его можно осмыслить целиком. Разброс логики синтаксического разбора по множеству классов значительно затрудняет восприятие синтаксического анализатора во всей его полноте.
2. Если синтаксический анализатор слишком велик для того, чтобы понять его целиком, то можно автоматически создать анализатор на основе про-

дукций. Все имеющиеся инструменты, известные автору этой книги, создают синтаксический анализатор в виде единственного класса или в виде основного класса с несколькими вспомогательными классами. Если существуют другие средства, создающие синтаксический анализатор в виде множества классов, то организация таких классов, скорее всего, будет отличаться от любой создаваемой вручную структуры. Это значит, что, если синтаксический анализатор организован вручную в виде множества классов, переход к автоматически создаваемому синтаксическому анализатору приведет к разрушению классов, так как они ссылаются на исчезнувший класс созданного вручную синтаксического анализатора. Создание синтаксического анализатора вручную в виде множества классов может затруднить переход к автоматически создаваемому синтаксическому анализатору.

Большая часть подклассов класса `TerminalToken` не содержит ни методов, ни переменных. Подклассы класса `TerminalToken`, которые должны содержать переменные, обычно имеют только одну переменную, значение которой представляет собой подстроку входных данных, которая сравнивается экземплярами класса. Поскольку подклассы класса `TerminalToken` содержат очень мало информации, большинство реализаций шаблона Little Language не утруждают себя использованием класса `TerminalToken` или его подклассов. Вместо этого они передают синтаксическому анализатору тип лексемы, опознанной лексическим анализатором, и соответствующую строку, полученную от лексического анализатора, не инкапсулируя в каком-либо объекте.

Синтаксические анализаторы создают синтаксические деревья снизу вверх. Корень дерева синтаксического разбора соответствует как минимум исходной программе со всеми ее потомками. По мере того как синтаксический анализатор выполняет синтаксический разбор своих входных данных, он создает небольшие синтаксические деревья. Распознавая все более и более длинные конструкции, он объединяет небольшие синтаксические деревья в более крупные, имеющие общий корень. По окончании синтаксического анализа получается только одно большое синтаксическое дерево.

Многие тонкости оптимизации и проектирования, очень важные для полновесных языков, не имеют никакого значения для малых языков. Суть в том, что способы, описанные в данном шаблоне, недостаточны для проектирования или реализации языков, подобных языку Java.

СЛЕДСТВИЯ

- Шаблон Little Language позволяет пользователям задавать различные комбинации операций. Почти всегда легче спроектировать и реализовать малый язык, чем GUI, обеспечивающий такую же гибкость и выразительность, какую предоставляет малый язык. Несмотря на это, большинство пользователей считает, что GUI более прост в использовании.
- Язык — это форма пользовательского интерфейса. Подобно работе с любым другим пользовательским интерфейсом, изучают, что с его помощью дела-

ется просто, а что не очень, и наблюдают за другими людьми, которые его применяют.

- Обычно класс `Parser` для малого языка реализуется посредством написания закрытых методов, которые в основном соответствуют нетерминальным лексемам. Такую схему легко понять, а грамматические изменения легко реализуются до тех пор, пока грамматика остается относительно небольшой. Если язык становится слишком громоздким, структура делается неуправляемой.

Для более значительных и предоставляющих полный сервис языков имеются более сложные технологии проектирования и реализации. Существуют инструменты, которые могут автоматически создавать классы `Parser` и `LexicalAnalizer` на основе продукций и правил лексического анализа. Другие инструменты помогают при создании и упрощении деревьев синтаксического разбора.

ПРИМЕНЕНИЕ В JAVA API

Подклассы класса `java.text.Format` используют шаблон Little Language. Конструкторам этих классов (явно или неявно) передается строка, содержащая описание формата на малом языке. Каждый подкласс имеет свой собственный малый язык для таких операций, как замена текста в сообщениях (`MessageFormat`), форматирование даты и времени (`DateFormat`) и форматирование десятичных чисел (`DecimalFormat`).

Ввиду простой структуры таких малых языков их синтаксические анализаторы создают не синтаксическое дерево, а массив объектов.

ПРИМЕР КОДА

В качестве первого примера приведем код лексического анализатора. Лексические правила языка словосочетаний достаточно похожи на лексические правила языка Java, поэтому класс `java.io.StreamTokenizer` может сделать большую часть работы. Это тот самый класс, который компилятор Java фирмы Sun использует для лексического анализа.

```
class LexicalAnalyzer {
   private StreamTokenizer input;
   private int lastToken;

   // Константы для идентификации типа последней лексемы.
   static final int INVALID_CHAR = -1;// Непредвиденный символ.
   static final int NO_TOKEN = 0;// Еще нет распознанных лексем.
   static final int OR = 1;
   static final int AMD = 2;
   static final int NEAR = 3;
```

```
static final int NOT = 4;
static final int WORD = 5;
static final int LEFT_PAREN = 6;
static final int RIGHT_PAREN = 7;
static final int QUOTED_STRING = 8;
static final int EOF = 9;

/**
 * Constructor
 * @param input Входной поток, подлежащий считыванию.
 */
LexicalAnalyzer(InputStream in) {
   input = new StreamTokenizer(in);
   input.resetSyntax();
   input.eolIsSignificant(false);
   input.wordChars('a', 'z');
   input.wordChars('A','Z');
   input.wordChars('0','9');
   input.wordChars('\u0000',' ');
   input.ordinaryChar('(');
   input.ordinaryChar(')');
   input.quoteChar('"');
} // constructor(InputStream)

/**
 * Возвращает строку, распознанную как лексема слова,
 * или тело строки, заключенной в кавычки.
 */
String getString() {
   return input.sval;
} // getString()

/**
 * Возвращает тип следующей лексемы. Для лексем слова
 * или строки в кавычках строка, представляемая лексемой,
 * может быть считана посредством вызова метода getString.
 */
int next Token() {
   int token;
   try {
      switch (input.next Token()) {
         case StreamTokenizer.TT_EOF:
```

```
                token = EOF;
                break;
             case StreamTokenizer.TT_WORD:
                if (input.sval.equalsIgnoreCase("or"))
                   token = OR;
                else if (input.sval.equalsIgnoreCase("and"))
                   token = AND;
                else if (input.sval.equalsIgnoreCase("near"))
                   token = NEAR;.
                else if (input.sval.equalsIgnoreCase("not"))
                   token = NOT;
                else
                   token = WORD;
                break;
             case '"';
                token = QUOTED_STRING;
                break;
             case '(':
                token = LEFT_PAREN;
                break;
             case ')':
                token = RIGHT_PAREN;
                break;
             default:
                token = INVALID_CHAR;
             break;
          } // switch
       } catch (IOException e) {
          // IOException считается концом файла.
          token = EOF;
       } // try
       return token;
    } // nextToken()
} // class LexicalAnalyzer
```

Хотя класс `LexicalAnalyzer` использует класс `StringTokenizer` для выполнения значительной части лексического анализа, он предоставляет свои собственные коды для указания типа распознанной им лексемы, что позволяет изменять реализацию класса `LexicalAnalyzer`, не затрагивая классы, использующие класс `LexicalAnalyzer`.

При реализации синтаксического анализатора используется способ, который называется *рекурсивным спуском*. Синтаксический анализатор, применяющий рекурсивный спуск, имеет методы, которые соответствуют нетерминальным лексемам, определяемым грамматическими продукциями. Эти методы вызывают друг друга, используя примерно тот же шаблон, согласно которому ссылаются друг на друга соответствующие грамматические продукции. Там, где есть рекурсия в грамматических продукциях, обычно имеется рекурсия и в методах. Одно важное исключение из этого правила относится к тому случаю, когда рекурсия является саморекурсией, выполняемой через крайнюю правую лексему продукции, например:

orCombination → *andCombination* **or** *orCombination*

Это можно пояснить так: метод саморекурсии создает метод, который выполняет саморекурсию, и это последнее, что он делает перед возвратом результата. Рекурсия такого вида представляет собой специальный случай, получивший название *концевой рекурсии*. Концевая рекурсия является специальным видом рекурсии, поскольку ее всегда можно преобразовать в цикл. Следующий код для класса `Parser` демонстрирует методы, соответствующие нетерминальным лексемам, которые определены при помощи саморекурсии. Эти методы реализуют саморекурсию, используя цикл.

```
public class Parser {
   private LexicalAnalyzer lexer; // Лексический анализатор.
   private int token;

   /**
    * Синтаксический анализ комбинации слов, прочитанных
    * из данного входного потока.
    * @param input
    *     Чтение комбинаций слов из этого InputStream.
    * @return Объект комбинации, который является корнем
    *         синтаксического дерева.
    */
   public Combination parse(InputStream input)
              throws SyntaxException{
      lexer = new LexicalAnalyzer(input);
      Combination c = orConbination();
      expect(LexicalAnalyzer.EOF);
      return c;
   } // parse(InputStream)

   private Combination orConbination(
              throws SyntaxException {
```

```
    Combination c = andCombination();
    while (token == LexicalAnalyzer.OR) {
      c = new OrCombination(c, andCombination());
    } // while
    return c;
  } // orCombination()

  private Combination andCombination()
            throws SyntaxException {
    Combination c = nearCombination();
    while (token == LexicalAnalyzer.AND) {
      c = new AndCombination(c, nearCombination());
    } // while
    return c;
  } // andCombination

  private Combination nearCombination()
            throws SyntaxException {
    Combination c = simpleConibination();
    while (token = LexicalAnalyzer.NEAR) {
      c = new NearCombination(c, simpleCombination());
    } // while
    return c;
  } // nearCombination()

  private Combination simpleCombination()
            throws SyntaxException {
    if (token == LexicalAnalyzer.LEFT_PAREN) {
      nextToken();
      Combination c = orCombination();
      expect(LexicalAnalyzer.RIGHT_PAREN) ;
      return c;
    } // if '('
    if (token == LexicalAnalyzer.NOT)
      return notWordCombination();
    else
      return wordCombination();
  } // simpleCombination()

  private Combination wordCombination()
            throws SyntaxException {
    if (token != LexicalAnalyzer.WORD
      && token != LexicalAnalyzer.QUOTED_STRING) {
```

```
            // Выводит сообщения об ошибке и генерирует исключения
            // SyntaxExeption.
            expect(LexicalAnalyzer.WORD);
        } // if
        Combination c = new WordCombination(lexer.getString());
        nextToken();
        return c;
    } // wordCombination()

    private Combination notWordCombination()
            throws SyntaxException {
        expect(LexicalAnalyzer.NOT);
        if (token != LexicalAnalyzer.WORD
            && token != LexicalAnalyzer.QUOTED_STRING) {
            // Выводит сообщения об ошибке и генерирует исключения
            // SyntaxExeption.
            expect (LexicalAnalyzer.WORD);
        } // if
        Combination c;
        c = new NotWordCombination(lexer.getString());
        nextToken();
        return c;
    } // notWordCombination()

    // Получает от лексического анализатора следующую лексему.
    private void nextToken() {
        token = lexer.nextToken();
    } // nextToken()
```

Остальная часть класса `Parser` — это метод под названием `expect` и вспомогательный метод метода `expect` — `tokenName`. Метод `expect` выдает сообщение об ошибке, если тип текущей терминальной лексемы не соответствует типу лексемы, переданной методу `expect`. Если тип лексемы соответствует ожидаемому, то метод `expect` считывает следующую лексему из лексического анализатора.

Большинство синтаксических анализаторов, использующих рекурсивный спуск, имеют метод, аналогичный `expect`. Он часто так и называется — `expect`.

```
    // Генерирует исключения, если текущая лексема не имеет
    // заданный тип.
    private void expect(int t) throws SyntaxException {
        if (token != t) {
```

```
            String msg = "found " + tokenName(token)
               + " when expecting " + tokenName(t);
               throw new SyntaxException(msg);
            } // if
            nextToken();
         } // expect(int)

         private String tokenName(int t) {
            String tname;
            switch (t) {
               case LexicalAnalyzer.OR:
                  tname = "OR";
                  break;
               case LexicalAnalyzer.AND:
                  tname = "AND";
                  break;
               case LexicalAnalyzer.NEAR:
                  tname = "NEAR";
                  break;
               case LexicalAnalyzer.NOT:
                  tname = "NOT";
                  break;
               case LexicalAnalyzer.WORD:
                  tname = "word";
                  break;
               case LexicalAnalyzer.LEFT_PAREN:
                  tname = "(";
                  break;
               case LexicalAnalyzer.RIGHT_PAREN:
                  tname = ")";
                  break;
               case LexicalAnalyzer.QUOTED_STRING:
                  tname = "quoted string";
                  break;
               case LexicalAnalyzer.EOF:
                  tname = "end of file";
                  break;
               default:
                  tname = "???";
                  break;
```

```
        } // switch
        return tname;
      } // tokenName(int)
    } // class Parser
```

Существует очевидная взаимосвязь между продукциями формальной грамматики и вышеприведенным кодом для класса `Parser`. Ввиду такой очевидной взаимосвязи может возникнуть большое искушение отказаться от написания формальной грамматики и определить малый язык просто при помощи кода. Игнорирование формальной грамматики — это, как правило, плохая идея в силу следующих причин.

- Без формальной грамматики не существует точного способа передачи определения языка другим пользователям, не заставляя их читать исходный код.
- По мере того как синтаксис языка становится более объемным и более сложным, таким же становится и синтаксический анализатор языка. Если синтаксический анализатор делается более сложным, код обрастает необходимыми деталями, и взаимосвязь между кодом и реализуемой им грамматикой становится менее очевидной.
- Как правило, языки со временем развиваются и приобретают новые свойства. Пытаясь внести изменения в язык, не имеющий формальной грамматики, можно столкнуться с трудностями при определении различий между изменением грамматики языка и изменением реализации этой грамматики.

Следующий фрагмент кода в этом примере — класс `Combination`, который является абстрактным суперклассом для всех объектов синтаксического дерева:

```
abstract class Combination {
  /**
   * Если данная строка содержит слова, которые нужны
   * этому объекту Combination, метод возвращает массив
   * целых чисел. В большинстве случаев в массиве
   * находятся значения смещений слов
   * в строке, которые соответствуют этой комбинации.
   * Однако если массив пустой, то все слова в строке
   * соответствуют комбинации. Если данная строка
   * не содержит слов, требуемых объектом Combination,
   * то этот метод возвращает null.
   */
  abstract int[] contains(String s) ;
} // class Combination
```

Очевидно, что методы класса `Combination` и его подклассов относятся в основном только к выполнению комбинаций. Здесь почти нет кода, связанного

с манипуляциями над объектами дерева синтаксического разбора. Некоторые более крупные языки требуют дополнительного анализа программы после проведения синтаксического разбора, с той целью, чтобы она могла быть выполняемой. Для таких языков дерево синтаксического разбора представляет собой промежуточную форму программы, отличную от ее выполняемой формы. Шаблон Little Language полагает, что язык достаточно прост для того, чтобы дерево синтаксического разбора можно было использовать двояко.

Приведем исходный код для класса `NotWordCombination`, который является простейшим подклассом класса `Combination`:

```
class NotWordCombination extends Combination {
  private String word;

  /**
   * constructor
   * @param word Слово в строке, которого требует эта
   * комбинация.
   */
  NotWordCombination(String word) {
    this.word = word;
  } // constructor(String)

  /**
   * Если данная строка содержит слово, необходимое этому
     объекту NotWordCombination,
   * то метод возвращает массив смещений,
   * соответствующих появлению слова в строке.
   * В противном случае этот метод возвращает null.
   */
  int[] contains(String s) {
    if (s.indexOf(word) >= 0)
      return null;
    return new int[0];
  } // contains(String)
} // class NotWordCombination
```

Класс `WordCombination` похож на класс `Combination`. Основное отличие состоит в том, что он содержит логику для возврата вектора смещений для всех случаев появления в данной строке слова, связанного с объектом `WordCombination`.

Подклассы класса `Combination`, представляющие логические операторы `OrCombination`, `AndCombination` и `NearCombination`, являются более сложными. Они отвечают за комбинирование результатов двух объектов-потомков

Combination. Приведем исходный код для AndCombination и OrCombination:

```
class AndCombination extends Combination {
   private Combination leftChild, rightChild;

   AndCombination(Combination leftChild,
                  Combination rightChild) {
      this.leftChild = leftChild;
      this.rightChild = rightChild;
   } // constructor(Combination, Combination)

   int[] contains(String s) {
      int[] leftResult = leftChild.contains(s);
      int[] rightResult = rightChild.contains(s);
      if (leftResult == null ||rightResult == null)
         return null;
      if (leftResult.length == 0)
         return rightResult;
      if (rightResult.length == 0)
         return leftResult;

      // Сортирует результаты с целью их сравнения и объединения.
      Sorter.sort(leftResult);
      Sorter.sort(rightResult);

      // Подсчитывает общие смещения с целью выяснения,
      // существуют ли общие смещения и сколько их.
      int conmonCount = 0;
      for(int l=0,r=0;
         l<leftResult.length && r<rightResult.length;){
         if (leftResult[l] < rightResult[r]) {
            l++;
         } else if (leftResult[l] > rightResult[r]) {
            r++;
         } else {
            commonCount++;
            l++;
            r++;
         } // if
      } // for
      if (commonCount == 0)
```

```
        // Объединяет общие результаты.
        int[] myResult = new int[commonCount];
        commonCount = 0;
        for (int 1=0,r=0;
           1<leftResult.length && r<rightResult.length;){
           if (leftResult[l] < rightResult[r]) {
              1++;
           } else if (leftResult[l] > rightResult[r]) {
              r++;
           } else {
              myResult[commonCount] = leftResult[l];
              commonCount++;
           1++;
           r++;
           } // if
        } // for
        return myResult;
     } // contains(String)
  } // class AndCombination

  class OrCombination extends Combination {
     private Combination leftChild, rightChild;

     /**
      * Constructor
      * @param left Левый потомок этого объекта.
      * @param right Правый потомок этого объекта.
      */
     OrCombination(Combination left. Combination right) {
        leftChild = left;
        rightChild = right;
     } // constructor(Combination, Combination)

     int[] contains(String s) {
        int[] leftResult = leftChild.contains(s);
        int[] rightResult = rightChild.contains(s);
        if (leftResult == null)
           return rightResult;
        if (rightResult == null)
           return leftResult;
        if (leftResult.length == 0)
```

```
            return leftResult;
         if (rightResult.length == 0)
            return rightResult;
         // Создает массив объединенных результатов.
         nt[] myResult
            = new int[leftResult.length + rightResult.length];
         System.arraycopy(leftResult, 0, myResult, 0,
                 leftResult.length);
         System.arraycopy(rightResult, 0, myResult,
                 leftResult.length,
                 rightResult.length);
         return myResult;
      } // contains(String)
   } // class OrCombination
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ LITTLE LANGUAGE

Composite. Дерево синтаксического разбора организовано при помощи шаблона Composite.

Visitor. Шаблон Visitor позволяет инкапсулировать в единственном классе логику выполнения несложных манипуляций с деревом синтаксического разбора.

Mediator (Посредник)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Шаблон Mediator использует один объект для согласования изменения состояний других объектов. Вместо распределения логики по разным классам он помещает логику, предназначенную для управления изменением состояний других объектов, в один класс-посредник. В результате получается более сцепленная реализация логики и более низкая связанность этих классов.

КОНТЕКСТ

Шаблон Mediator решает проблему, которая часто встречается при использовании диалоговых окон. Предположим, что для задания информации для резервирования банкетного зала в отеле необходимо реализовать диалоговое окно (рис. 8.10).

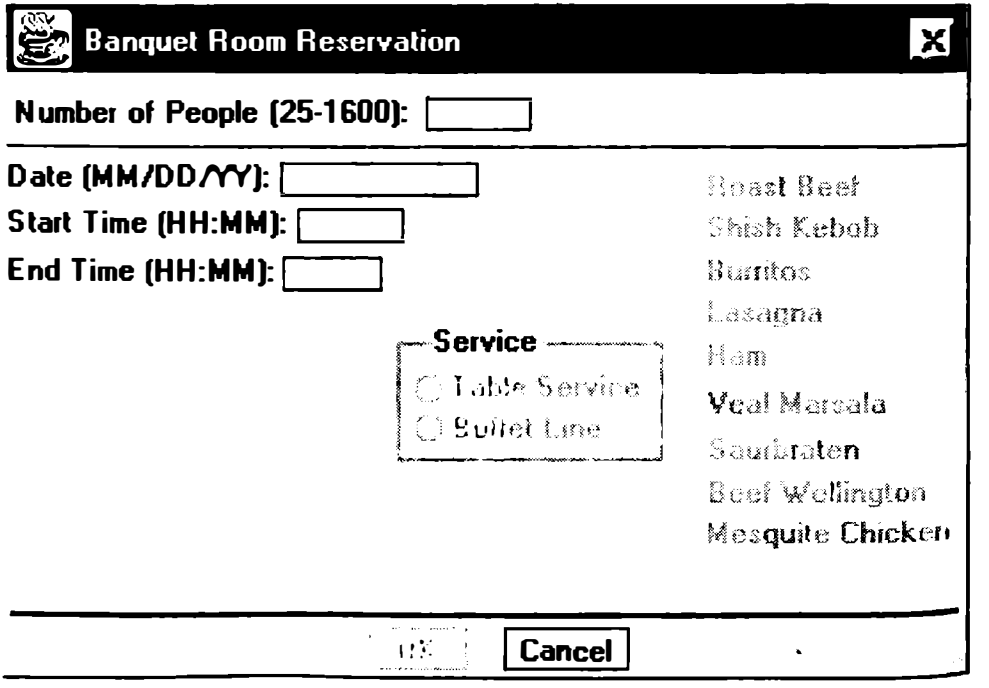


Рис. 8.10. Диалоговое окно для заказа банкетного зала

Назначение этого диалогового окна состоит в том, чтобы предоставить необходимую информацию для резервирования банкетного зала в отеле. Условия диалогового окна соответствуют зависимостям между объектами диалога.

- При первом появлении диалогового окна доступны только поля Number of People (Количество человек) и кнопка отмены (Cancel). До тех пор пока в это поле не будет введено число в диапазоне от 25 до 1600, остальная часть диалога недоступна. После ввода количества человек становятся доступными поля Date (Дата), Start Time (Время начала) и End Time (Время конца), но они позволяют вводить только те значения времени, когда зал нужных размеров не занят. Доступны также кнопки-переключатели. Последующие изменения, вводимые в поле Number of People, делают доступными другие поля и кнопки-переключатели.
- Время начала должно быть раньше времени конца.
- Когда пользователь ввел данные в поля времени и даты и выбрал кнопку-переключатель, появляется список блюд. Заказанные дата, время и вид обслуживания определяют блюда, входящие в список. Некоторые блюда являются сезонными, и отель предлагает их только в течение некоторого периода времени года. Блюда для завтрака будут находиться в списке, предлагаемом только для утренних банкетов. Некоторые блюда не годятся для шведских столов, а должны подаваться официантами.
- Если выбрано хотя бы одно блюдо и текстовые поля содержат правильные данные, доступна кнопка OK.

Если каждый объект диалогового окна отвечает за те зависимости, с которыми он связан, в результате получится сильно связанный набор объектов, имеющих слабое сцепление (рис. 8.11).

Для упрощения этой диаграммы на ней не указаны имена ассоциаций, ролей и индикаторы множественности. Целью диаграммы является отображение всех связей. Очевидно, что каждый объект участвует по крайней мере в двух зависимостях, а для некоторых объектов число зависимостей доходит до пяти. Большая часть времени, затрачиваемого на реализацию диалогового окна, пойдет на кодирование 15 связей зависимости.

Логика управления зависимостью распределена по восьми объектам, поэтому реализация диалогового окна сопряжена с немалыми трудностями. При работе над диалоговым окном программист будет видеть только небольшую часть управления зависимостями. Ему будет сложно понять детали управления зависимостями в целом, и он не будет тратить на это время. Если программисты, поддерживая код, не имеют полного представления обо всех деталях, сопровождение отнимает много времени, и часто его качество невысоко.

Очевидно, что необходима реорганизация этих объектов, позволяющая сделать количество связей минимальным и объединить все управление зависимостями в одном сцепленном объекте. Такое усовершенствование экономит время программиста и позволяет получить более надежный код. Именно для этой цели

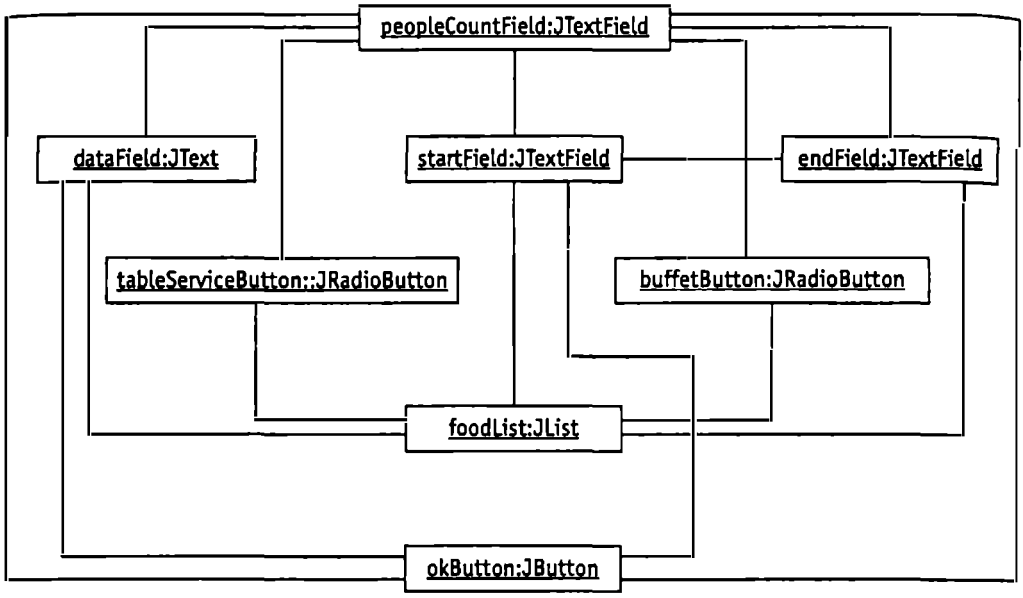


Рис. 8.11. Децентрализованное управление зависимостями

предназначен шаблон Mediator. При использовании этого шаблона объект не должен самостоятельно управлять своими связями с другими объектами, вместо этого применяется другой объект, который берет на себя все управление зависимостями. При такой организации каждый объект имеет только одну связь зависимости.

На рис. 8.12 представлены объекты диалога, организованные при помощи дополнительного объекта, предназначенного для централизованного управления зависимостями.

Помимо упрощения реализации и поддержки кода, представленный на рис. 8.12 проект более понятен, чем показанный на рис. 8.11.

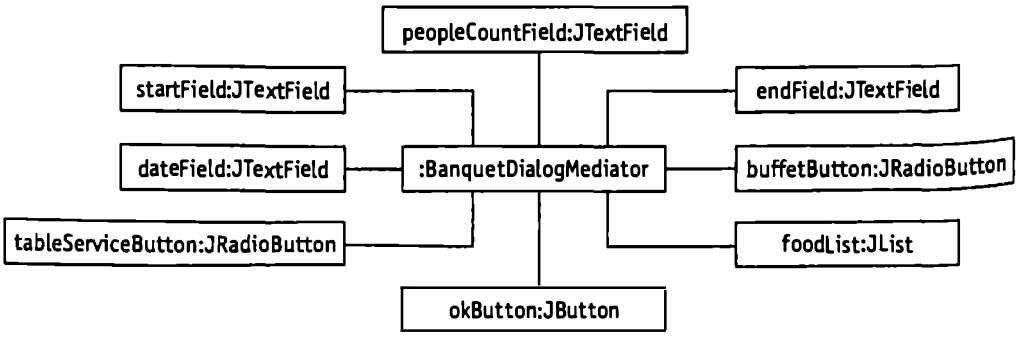


Рис. 8.12. Централизованное управление зависимостями

МОТИВЫ

☺ Сушествует набор связанных объектов, и почти все объекты участвуют во множестве отношений зависимостей.

☺ При определении подклассов предполагается участие отдельных объектов в отношениях зависимости.

☺ Классы трудно использовать многократно, так как их функция усложнена отношениями зависимостей.

РЕШЕНИЕ

На диаграмме взаимодействия, представленной на рис. 8.13, показан обший случай участия классов и интерфейсов в шаблоне Mediator.

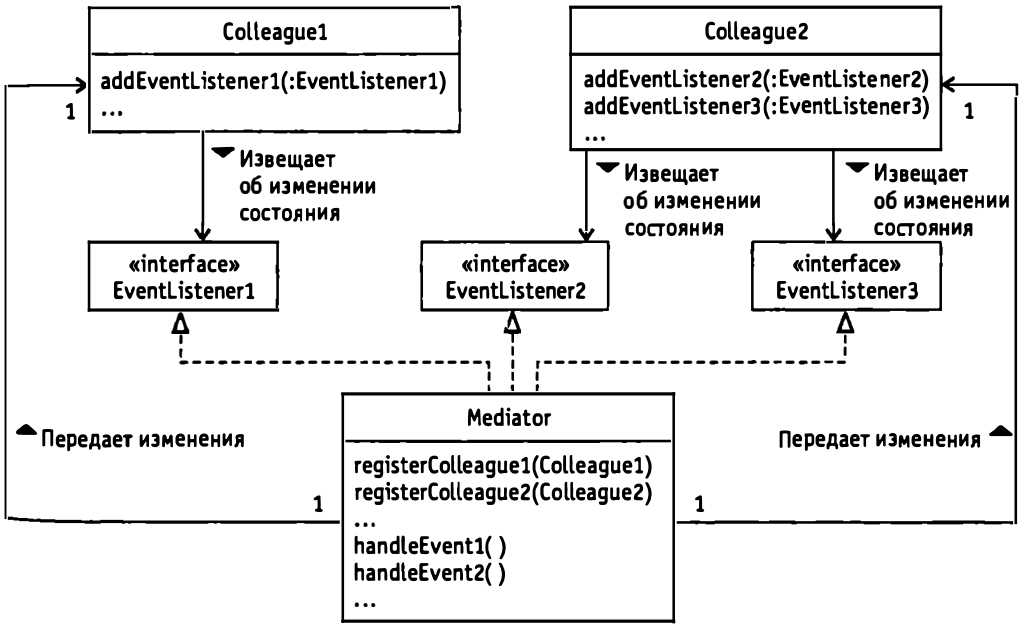


Рис. 8.13. Классы шаблона Mediator

Опишем роли, исполняемые этими классами и интерфейсами.

Colleague1, Colleague2 и т.д. Экземпляры классов в этой роли имеют зависимости, связанные с состояниями:

- один тип зависимости требует от объекта получить одобрение со стороны других объектов на выполнение специальных изменений состояния;
- другой тип зависимости требует от объекта оповешения иных объектов о том, что он выполнил изменение состояния.

Оба типа зависимостей обрабатываются похожим образом. Экземпляры классов `Colleague1`, `Colleague2` и т.д. связаны с объектом `Mediator`. Если им нужно оповестить другие объекты об изменении состояния, они вызывают метод объекта `Mediator`. Метод объекта `Mediator` делает все остальное.

EventListener1, EventListener2 и т.д. Интерфейсы в этой роли дают возможность классам `Colleague1`, `Colleague2` и т.д. достичь высокой степени многократного использования, поскольку позволяют классам не знать, что они работают с объектом `Mediator`. Каждый из этих интерфейсов определяет методы, связанные с событием определенного вида. Чтобы обеспечить извещения о состоянии, объекты `Colleague` вызывают соответствующий метод нужного интерфейса, ничего не зная о классе объекта `Mediator`, реализующем этот метод.

Mediator. Экземпляры классов в роли `Mediator` имеют логику для обработки извещений о состоянии, полученных от объектов `Colleague1`, `Colleague2` и т.д. Классы `Mediator` реализуют один или несколько интерфейсов `EventListener`. Объекты `Colleague1`, `Colleague2` вызывают методы, объявленные интерфейсами `EventListener` с целью проинформировать объект `Mediator` об изменении состояния. Затем объект `Mediator` выполняет соответствующую логику: для извещений о предполагаемых изменениях состояния логика обычно содержит одобрение или неодобрение изменения, для извещений о выполненных изменениях состояния она обычно передает такие извещения другим объектам.

Классы `Mediator` имеют методы, которые могут быть вызваны для их связи с объектами `Colleague1`, `Colleague2` и т.д. Эти методы указаны на диаграмме как `registerColleague1`, `registerColleague2` и т.д. Они передаются соответствующему объекту `Colleague` и обычно вызывают один или несколько его методов `add...Listener` с целью проинформировать объект `Colleague` о том, что он должен известить объект `Mediator` об изменении состояния.

Средства, предоставляемые объектами `Colleague1`, `Colleague2` и т.д. и позволяющие другим объектам выразить свою заинтересованность в изменении состояния, и механизм передачи извещений о таких изменениях обычно согласуются с моделью делегирования событий в языке Java.

РЕАЛИЗАЦИЯ

Во многих случаях один объект отвечает за создание всех объектов `Colleague` и соответствующего объекта `Mediator`. Такой объект обычно выполняет роль контейнера для создаваемых им объектов. Если существует единственный объект, отвечающий за создание всех объектов `Colleague` и их объекта `Mediator`, то класс `Mediator`, как правило, объявляется как закрытый член этого класса. Ограничивая видимость класса `Mediator`, повышают надежность программы.

При реализации шаблона `Mediator` нужно принять некоторые решения. Одно из таких решений заключается в том, будет ли объект `Mediator` поддерживать свою собственную внутреннюю модель состояния объектов `Colleague` или он предпочтет считывать состояние каждого объекта всякий раз, когда потребуется информация о состоянии объекта.

Если применяется первый подход, то объект Mediator начинает с того, что определяет первоначальное состояние всех объектов Colleague, за которые отвечает. Он будет иметь переменную экземпляра для каждого объекта Colleague. Объект Mediator задает начальное значение каждой из этих переменных экземпляра, которое, как он считает, должно быть начальным состоянием соответствующего объекта Colleague. Когда объект Colleague оповещает объект Mediator об изменении своего состояния, Mediator изменяет значения своих переменных экземпляра, приводя их в соответствие с новым состоянием. После обновления своих переменных экземпляра объект Mediator использует значения этих переменных для принятия нужного решения.

Если применяется второй подход, то объект Mediator не пытается моделировать состояние объектов Colleague при помощи своих переменных экземпляра. Вместо этого, когда объект Colleague оповещает его об изменении состояния, объект Mediator принимает нужные решения, считывая состояние каждого объекта Colleague, на основании которого он должен принимать эти решения.

Когда программисты впервые пытаются реализовать шаблон Mediator, они чаще всего выбирают первый подход. Однако в большинстве случаев второй подход является более удачным вариантом.

Недостаток первого механизма состоит в том, что объект Mediator может иметь неверную информацию о состоянии объекта Colleague. Чтобы точно моделировать состояние объектов Colleague, объект Mediator может испытывать потребность в дополнительном коде для имитации логики объектов Colleague. Если позднее программист из группы сопровождения изменит один из классов Colleague, такое изменение может разрушить класс Mediator.

Преимущество второго подхода заключается в его простоте. Объект Mediator никогда не будет иметь неправильную информацию о состоянии объекта Colleague. Поэтому реализация и поддержка объекта Mediator не вызывает затруднений. Однако если чтение полного состояния объектов Colleague требует слишком много времени, то может быть более практичным такой подход, при котором объект Mediator моделирует состояние объектов Colleague.

СЛЕДСТВИЯ

☺ Почти все сложности, связанные с управлением зависимостями, переходят от разных объектов к объекту Mediator. При этом упрощается реализация и поддержка разных объектов.

☺ Помещение всей логики зависимостей классов в один класс-посредник помогает программистам, осуществляющим сопровождение, находить зависимости.

☺ Использование объекта Mediator обычно приводит к меньшему количеству ветвей выполнения кода. Это значит, что требуется меньше усилий для полного тестирования, поскольку должно проверяться меньше случаев. Влияние меньшего количества ветвей выполнения кода на тестирование подробно рассматривается при описании шаблона White Box Testing в книге [Grand99].

- Использование объекта `Mediator` обычно означает отсутствие необходимости создания подклассов классов `Colleague` только для реализации управления их зависимостями.
- Классы `Colleague` являются в большей степени многократно используемыми, так как на их основную функциональность не влияет код, управляющий зависимостями. Он обычно имеет тенденцию зависеть от приложения.

  Размещение всей логики зависимостей набора связанных объектов в одном месте упрощает понимание этой логики зависимостей. Если класс `Mediator` получается слишком большим, то разбиение его на мелкие части может привести к тому, что он станет менее понятным.
- Код для управления зависимостями обычно зависит от приложения, поэтому классы `Mediator`, как правило, не являются многократно используемыми.

ПРИМЕНЕНИЕ В JAVA API

Примеры шаблона Mediator, которые можно найти в Java API, немного отличаются от тех посредников, которые, скорее всего, придется кодировать. Причина в том, что код классов `Mediator` обычно зависит от приложения, а классы Java API от приложения не зависят.

GUI на основе языка Java может быть создан, главным образом, из объектов, которые являются экземплярами подклассов класса `java.awt.swing.JComponent`. Объекты `JComponent` используют экземпляр подкласса класса `java.awt.swing.FocusManager` в качестве посредника. Если объекты `JComponent` связаны с объектом `FocusManager` (а они обычно связаны), то они вызывают его метод `processKeyEvent` при получении `KeyEvent`. Задача объекта `FocusManager` состоит в том, чтобы распознать нажатия клавиш, которые должны активизировать определенный объект `JComponent`, что он и делает.

Способ использования объектами `JComponent` объекта `FocusManager` отличается от описанного в шаблоне Mediator ввиду следующих двух причин:

1. Объекты `JComponent` передают ключевые события от нажатия клавиш только объектам `FocusManager` . Большая часть классов `Mediator`, которые придется писать, должны будут управлять событиями нескольких видов.
2. Объекты `JComponent` не обращаются к объектам `FocusManager` через интерфейс. Они прямо ссылаются на класс `FocusManager`. Если объекты `JComponent` ссылаются на объекты `FocusManager` через интерфейс, структура является более гибкой. Видимо, проектировщики Java API посчитали здесь такую гибкость ненужной, поскольку взаимодействие между объектами `JComponent` и объектами `FocusManager` осуществляется на низком уровне.

ПРИМЕР КОДА

Пример кода для шаблона Mediator — это код для объекта-посредника, который используется в диалоговом окне, рассматриваемом в разделе «Контекст». В этом примере отражена суть шаблона Mediator — возложить ответственность за всю сложность управления событиями на классы `Mediator`.

Класс-посредник реализуется как закрытый внутренний класс класса диалога под названием `BanquetMediator`:

```
private class BanquetMediator {
   private JButton okButton;
   private JTextComponent dateField;
   private JTextComponent startField;
...
```

Как было показано выше, класс `BanquetMediator` имеет закрытые переменные экземпляра, используемые им для ссылки на объекты GUI, которые диалоговое окно объявляет вместе с ним. Класс `BanquetMediator` не реализует каких-либо интерфейсов `EventListener`, позволяющих ему получать события от зарегистрированных объектов GUI. Вместо этого для получения таких событий он использует объекты адаптера. Ниже приводятся две основные причины, почему класс `BanquetMediator` использует классы-адаптеры для получения событий.

- Класс `BanquetMediator` может гарантировать, что только те объекты GUI, которые, как ожидается, должны отправлять события объекту `BanquetMediator`, действительно могут это сделать. При этом повышается надежность класса `BanquetMediator`. С этой целью объекты `BanquetMediator` делают свои методы управления событиями доступными только для зарегистрированных объектов GUI. Все методы управления событиями класса `BanquetMediator` объявляются как закрытые. Объекты-адаптеры являются экземплярами закрытого или анонимного внутреннего класса, который может вызывать внутренние методы класса `BanquetMediator`. Поскольку классы адаптера являются закрытыми или анонимными, их экземпляры могут создаваться только классом `BanquetMediator`. Экземпляры классов адаптера предоставляются только зарегистрированным объектам GUI.
- Использование объектов-адаптеров для обработки событий от разных объектов GUI освобождает класс `BanquetMediator` от необходимости определять, от какого именно объекта GUI поступило событие. Класс `BanquetMediator` объявляет закрытые или анонимные классы-адаптеры и использует их для получения событий только от объектов, играющих определенную роль. Объявляя дополнительные классы, класс `BanquetMediator` освобождает свои классы-адаптеры от необходимости выбора характера поведения, зависящего от источника события.

Примеры таких классов-адаптеров приводятся ниже в листинге. Анонимные классы-адаптеры используются для обработки событий, специфичных для каждо-

го источника события. Закрытые именованные классы-адаптеры применяются для обработки событий, поведение которых не зависит от источника событий. Класс `BanquetMediator` объявляет переменные экземпляра, ссылающиеся на единственный создаваемый им экземпляр своих закрытых адаптерных классов-адаптеров.

```
    private ItemAdapter itemAdapter = new ItemAdapter();
...
```

Конструктор класса `BanquetMediator` объявляет и инстанциирует анонимный класс-адаптер, обрабатывающий событие, которое возникает в том случае, если диалог открыт. Адаптер вызывает метод класса `BanquetMediator`, отвечающий за инициализацию начальных состояний зарегистрированных компонентов GUI.

```
BanquetMediator() {
    WindowAdapter windowAdapter = new WindowAdapter() {
      public void windowOpened(WindowEvent e) {
        initialState();
      } // windowOpened(WindowEvent)
    };
    BanquetReservationDialog enclosingDialog;
    enclosingDialog = BanquetReservationDialog.this;
    enclosingDialog.addWindowListener(windowAdapter);
  } // Constructor()
...
```

Класс `ItemAdapter` представляет собой закрытый класс-адаптер, который определяется и используется классом `BanquetMediator` для обработки событий от обеих кнопок-переключателей диалогового окна. Когда объект `ItemAdapter` получает `ItemEvent`, он вызывает метод `enforceInvariants` класса `BanquetMediator`. Этот метод является для класса `BanquetMediator` основным. Метод `enforceInvariants` активизирует все инвариантные отношения между компонентами диалога. Он вызывается в ответ на получение события от всех компонентов GUI-диалога.

```
private class ItemAdapter implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
        enforceInvariants();
      } // itemStateChanged(ItemEvent)
} // class ItemAdapter
```

Метод, отвечающий за появление объекта кнопки OK весьма прост, так как класс `BanquetMediator` не отвечает за обработку событий от этой кнопки. Он отвечает только за то, должна ли быть доступна или недоступна кнопка OK.

```
public void registerOkButton(JButton ok) {
        okButton = ok;
    } // registerOkButton(JButton)
```

Методы регистрации других объектов GUI более сложны, так как они затрагивают управление специальными событиями для объектов, зарегистрированных в определенной роли. Управление специальными событиями необходимо для того, чтобы проверить содержимое отдельных объектов GUI. Следующий метод регистрации более типичен. Он регистрирует поле, предназначенное для ввода количества участников предстоящего банкета.

```
public
void registerPeopleCountField(final JTextComponent field) {
        peopleCountField = field;
        DocumentAdapter docAdapter
           = new DocumentAdapter() {
           protected void parseDocument() {
              int count = PEOPLE_COUNT_DEFAULT;
              try {
                 String countText = field.getText();
                 count = Integer.parseInt(countText);
              } catch (NumberFormatException e) {
              }
              if (MIN_PEOPLE<=count
                 && count<=MAX_PEOPLE )
              peopleCount = count;
              else
              peopleCount = PEOPLE_COUNT_DEFAULT;
              } // parseDocument()
           };
        Document doc = field.getDocument();
        doc.addDocumentListener(docAdapter);
    } // registerPeopleCountField(JTextComponent)
```

Этот метод регистрации создает анонимный объект-адаптер, который не просто вызывает метод `enforceInvariants`. Об этом позаботился анонимный суперкласс-адаптер. Перед тем как вызвать метод `enforceInvariants` объекта `BanquetMediator`, суперкласс вызывает свой собственный метод `parseDocument`. Класс анонимного адаптера его замещает, устанавливая переменную экземпляра `peopleCount` объекта `BanquetMediator`. Если поле, соответствующее количеству человек, намеревающихся посетить банкет, содержит правильное значение, то адаптер это значение записывает в `peopleCount`. В противном случае в `peopleCount` задается специальное значение, информирующее метод `enforceInvariant` о том, что в поле количества человек, намеревающихся посетить банкет, введена неправильная величина.

Методы регистрации других текстовых полей работают аналогичным образом. Они создают объект-адаптер, который проверяет правильность введенного в поле значения, устанавливает переменную экземпляра в некоторое значение и затем вызывает метод `enforceInvariants`.

Метод `enforceInvariants` может изменять состояние некоторых компонентов GUI, чтобы заставить их удовлетворять некоторым инвариантным отношениям. При этом компоненты генерируют события. Объект `BanquetMediator` при помощи своих адаптеров получает некоторые из этих событий. Когда компонент GUI отвечает на одно из изменений состояния, сделанных методом `enforceInvariants`, и передает событие одному из объектов-адаптеров `BanquetMediator`, он рекурсивно вызывает метод `enforceInvariants`. Чтобы избежать бесконечной рекурсии, класс `BanquetMediator` использует флаг, указывающий на рекурсивные обращения к методу `enforceInvariants`.

```
private boolean busy = false;
...
     private void enforceInvariants() {
        if (busy)
           return;
        busy = true;
        protectedEnforceInvariants();
        busy = false;
     } // enforceInvariants()
```

Как видно, метод `enforceInvariants` активизирует инвариантные отношения непрямым образом. Если он вызывается рекурсивно, происходит немедленный возврат этого метода. В противном случае он вызывает метод `protectedEnforceInvariants`.

Метод `enforceInvariants` определяет рекурсивные обращения так: сначала проверяет, а затем устанавливает значение переменной `busy`. Поскольку модель управления событием в языке Java гарантирует синхронную передачу событий, метод `enforceInvariants` не должен быть вызван для управления некоторым событием в то время, как он все еще обрабатывает другое событие. В подобной ситуации метод должен быть синхронизирован в соответствии с семантикой обрабатываемых им событий.

Приведем инвариантные отношения, активизируемые методом `protectedEnforceInvariants`.

- Поля Date, Start Time и End Time разрешены тогда и только тогда, когда поле Number of People содержит разрешенное значение.
- Если кнопки-переключатели запрещены, их состояние не позволяет их выбирать.
- Список блюд доступен тогда и только тогда, когда доступны поля Data, Start Time и End Time и выбран переключатель Buffet (Шведский стол) или Table

(Обслуживание за столиками). Правильное время окончания банкета должно быть по крайней мере на один час позднее, чем время начала.

- Кнопка OK доступна тогда и только тогда, когда доступен список блюд и из списка выбрано хотя бы одно блюдо.

```
private void protectedEnforceInvariants() {
        // Если задано количество человек, переменная enable
        // должна иметь значение true.
        boolean enable
           = (peopleCount != PEOPLE_COUNT_DEFAULT);

        // Поля Data, Start Time, End Time, переключатели Buffet,
        // Table доступны тогда и только тогда, когда в поле,
        // предназначенном для количества человек,
        // содержится разрешенное значение.
        dateField.setEnabled(enable);
        startField.setEnabled(enable);
        endField.setEnabled(enable);
        buffetButton.setEnabled(enable);
        tableServiceButton.setEnabled(enable);
        if (enable) {
           // Список блюд доступен тогда и только тогда,
           // когда поля даты, времени или кнопки-переключатели
           // доступны, и время окончания по крайней мере
           // на один час больше времени начала, и выбран
           // переключатель Buffet или Table.
           enable = (buffetButton.isSelected()
              || tableServiceButton.isSelected()) ;
           foodList.setEnabled(enable
              && endAtLeastOneHourAfterStart()) ;
        } else {
           // Если поля даты или времени или кнопки-переключатели
           // недоступны, то список блюд тоже должен быть
           // недоступен.
           foodList.setEnabled(false);
           // Кнопки-переключатели недоступны, поэтому они не
           // могут быть выбраны.
           buffetButton.setSelected(false);
           tableServiceButton.setSelected(false);
        } // Если доступны, то
        okButton.setEnabled(foodList.isEnabled()
             && foodList.getMinSelectionIndex()>-1);
     } // protectedEnforceInvariants()
```

Классы-посредники часто имеют внутренние вспомогательные методы, которые дополняют логику основного метода активизации инварианта. Размещение некоторой части логики во вспомогательных методах способствует тому, что размеры основного метода активизации инварианта позволяют легко управлять им. Следующий метод возвращает true, если поля Date, Start Time и End Time содержат допустимые значения и время в поле End Time как минимум на один час больше времени в поле Start Time.

```
private boolean endAtLeastOneHourAfterStart() {
        Calendar startCalendar = getStartCalendar();
        if (startCalendar == null)
          return false;
        Calendar endCalendar = getEndCalendar();
        if (endCalendar == null)
          return false;
        startCalendar.add(Calendar.MINUTE, 59) ;
        return getEndCalendar().after(startCalendar);
} // endAtLeastOneHourAfterStart()
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ MEDIATOR

Adapter. Классы `Mediator` часто используют объекты-адаптеры для получения извещений об изменении состояния.

Interface. Шаблон Mediator использует шаблон Interface с целью поддержки независимости классов `Colleague` от класса `Mediator`.

Low Coupling/High Cohesion. Шаблон Mediator — это хороший пример исключения из тех рекомендаций, которые выдаются шаблоном Low Coupling/High Cohesion (описанным в книге [Grand99]).

Observer. Шаблон Observer представляет собой значительную часть модели делегирования событий в языке Java. Если необходимо использовать шаблон Mediator в таком контексте, в котором, по мнению программиста, модель делегирования событий в языке Java не может быть использована, то можно использовать для замены шаблон Observer.

Controller. Шаблон Controller (описанный в книге [Grand99]) позволяет определить, какой объект должен обрабатывать внешнее событие. Шаблон Mediator помогает реализовать обработку событий.

White Box Testing. Использование шаблона Mediator приводит в результате к меньшему количеству ветвей выполнения кода. Это значит, что требуется меньше усилий для тестирования программы при помощи шаблона White Box Testing, описанного в книге [Grand99].

Snapshot (Моментальный снимок)

Этот шаблон частично основан на шаблоне Memento, описанном в работе [GoF95].

СИНОПСИС

Записывает моментальный снимок состояния объекта таким образом, чтобы это состояние можно было восстановить позднее. Объект, который инициирует запись моментального снимка или восстановление состояния, может не иметь информации об этом состоянии. Он должен знать только, что объект, состояние которого он записывает или восстанавливает, реализует определенный интерфейс.

КОНТЕКСТ

Предположим, что создается программа для ролевой игры. Подробности игры не имеют значения, важно только, что она предназначена для одного игрока. Во время этой игры пользователь заставляет героя вступать во взаимодействие с различными действующими лицами, контролируемыми компьютером, и объектами, смоделированными компьютером. Программа может завершаться смертью героя, управляемого игроком.

Среди многочисленных свойств игры есть две характеристики, которые предполагают сохранение и восстановление состояния игры. Эти свойства очень необходимы программе, поскольку, если играть в такую игру до ее полного завершения, на это потребуется несколько дней непрерывной игры.

- Чтобы позволить игроку делать многочисленные перерывы во время игры, должна существовать возможность сохранения состояния игры в файле, чтобы позднее ее можно было продолжить.
- Чтобы довести игру до конца, пользователь должен успешно провести своего героя через множество приключений. Если герой, руководимый игроком, умирает, игрок имеет возможность начать игру с самого начала. Но такой вариант не всегда является наилучшим, поскольку игрок уже хорошо освоил игру и много раз проходил все предыдущие этапы. Лучше, чтобы программа предложила игроку вариант возобновления игры с момента, предшествующего смерти героя.

  С этой целью сохраняется часть состояния игры, включающая накопленный ранее опыт героя и запись некоторых его достижений. Частичное сохранение состояния будет производиться в тот момент, когда герой выполнит какое-то важное задание. В ходе игры такие контрольные точки становятся частью полного состояния игры, они должны сохраняться при записи на диск остальной части состояния.

Хотя в игре участвует множество классов, рассмотрим только некоторые из них, которые несут общую ответственность за создание моментальных снимков состояния игры (рис. 8.14).

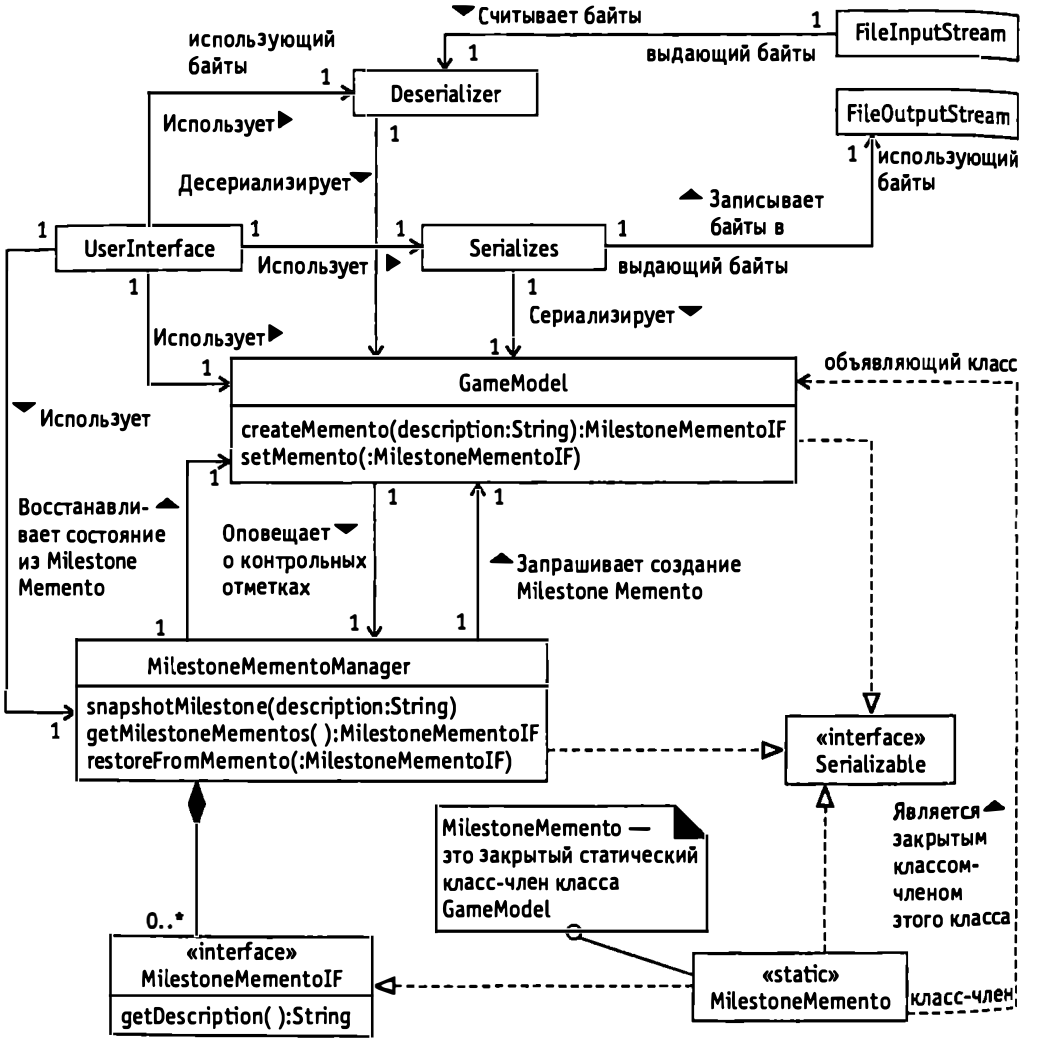


Рис. 8.14. Классы моментального снимка игры

Данные классы принимают участие в двух различных механизмах:

- сохранения части состояния игры, когда герой достигает контрольной отметки;
- сохранения и восстановления всей игры целиком.

Классы `UserInterface` и `GameModel` участвуют в обоих механизмах.

UserInterface. Все предпринимаемые игроком действия проходят через класс `UserInterface`.

Класс `UserInterface` передает бóльшую часть действий, инициируемых игроком, экземпляру класса `GameModel`. Однако в процессе работы с инициируемыми игроком моментальными снимками игры задействованы и другие подходы, которые будут рассмотрены ниже.

GameModel. Класс `GameModel` отвечает за поддержание состояния программы во время сеанса игры. Класс `UserInterface` оповещает экземпляр класса `GameModel`, если игрок делает что-то связанное с игрой. Экземпляр класса `GameModel` определяет последствия этого действия, соответствующим образом изменяет состояние игры и оповещает пользовательский интерфейс. Кроме того, экземпляр класса `GameModel` может сам инициировать некоторые действия. Он всегда сообщает о последствиях любого инициированного им действия пользовательскому интерфейсу, но не всегда — о самом действии.

Участие класса `UserInterface` в процессе создания моментального снимка состоит в том, что он инициирует один вид моментального снимка и оба вида восстановления. Класс `GameModel` отвечает за состояние игры, поэтому он должен участвовать в любой операции, манипулирующей состоянием игры.

Опишем другие классы и интерфейсы, принимающие участие в сохранении и восстановлении частичного состояния.

MilestoneMemento. Класс `MilestoneMemento` — это закрытый класс, определяемый классом `GameModel`. Объект `GameModel` создает экземпляры класса `MilestoneMemento`, содержащие копии значений, которые определяют сохраняемое частичное состояние. Имея объект `MilestoneMemento`, объект `GameModel` может восстановить свое предыдущее состояние, содержащееся в объекте `MilestoneMemento`.

MilestoneMementoIF. Этот интерфейс является открытым и реализуется классом `MilestoneMemento`. За пределами класса `GameModel` доступ к экземплярам класса `MilestoneMemento` может быть осуществлен только как к экземплярам класса `Object` или через интерфейс `MilestoneMementoIF`. Ни один режим доступа не позволяет объекту обращаться к информации о состоянии, инкапсулированной в объектах `MilestoneMemento`.

MilestoneMementoManager. Класс `MilestoneMementoManager` вносит свой вклад в процесс принятия решения о создании объектов `MilestoneMemento`. Он также управляет их использованием после того, как они были созданы.

Приведем описание записи частичного состояния игры, происходящего после того, как управляемый игроком герой достиг контрольной метки.

1. Объект `GameModel` принимает состояние, указывающее на то, что управляемый игроком герой достиг одной из главных контрольных меток игры.
2. Существует объект `MilestoneMementoManager`, связанный с каждым объектом `GameModel`. Когда объект `GameModel` принимает состояние, соответст-

вующее моменту достижения контрольной отметки, он вызывает метод snapshotMilestone связанного с ним объекта MilestoneMementoManager. Метод snapshotMilestone передает методу строку, в которой содержится описание контрольной метки. Управляемый игроком герой мог ранее уже быть у этой контрольной метки, умереть и затем возвратиться к предыдущей контрольной метке. Если объект MilestoneMemento для некоторой контрольной метки уже существует, то для этой контрольной метки не должен создаваться другой объект MilestoneMemento.

3. Объект MilestoneMementoManager определяет, существует ли уже объект MilestoneMemento для контрольной отметки, сравнивая строку описания, переданную его методу snapshotMilestone, с описаниями уже существующих объектов MilestoneMemento. Если объект MilestoneMemento с данным описанием уже существует, то метод snapshotMilestone не предпринимает никаких дополнительных действий.

4. Если объект MilestoneMementoManager определяет, что для контрольной отметки еще не существует объект MilestoneMemento, то объект MilestoneMementoManager инициирует создание объекта MilestoneMemento для записи частичного состояния игры в данный момент. При этом он вызывает метод createMemento объекта GameModel, передавая ему то же самое описание, которое он передавал объекту MilestoneMementoManager.

5. Метод createMemento возвращает вновь созданный объект MilestoneMemento, который объект MilestoneMementoManager добавляет в свою коллекцию объектов MilestoneMementoIF.

Если руководимый игроком герой умирает, объект UserInterface предлагает игроку снова включиться в игру не с самого начала, а с последней пройденной контрольной отметки. Он предлагает игроку сделать выбор из списка контрольных отметок, вызывая метод getMilestoneMementos объекта MilestoneMementoManager. Этот метод возвращает массив объектов MilestoneMemento, который составляет коллекцию, собранную объектом MilestoneMementoManager.

Если игрок указывает, что он хочет, чтобы герой возобновил игру от одной из ранее пройденных контрольных отметок, объект UserInterface передает соответствующий объект MilestoneMemento методу restoreFromMemento объекта MilestoneMementoManager. Этот метод в свою очередь вызывает метод setMemento объекта GameModel, передавая ему выбранный объект MilestoneMemento. Объект GameModel восстанавливает свое состояние, используя информацию, находящуюся в этом объекте MilestoneMemento.

При использовании другого механизма моментального снимка в файле сохраняется полное состояние игры, включая объект MilestoneMementoManager и его коллекцию объектов MilestoneMemento. Этот механизм основан на средстве сериализации языка Java.

В процессе сохранения (восстановления) в файл (из файла) моментального снимка полного состояния игры участвуют следующие классы.

Serializer. Класс `Serializer` отвечает за сериализацию объекта `GameModel`. Он копирует в файл (представляя в виде байтового потока) информацию о состоянии объекта `GameModel` и других объектов, на которые он ссылается и которые являются частью состояния игры.

FileOutputStream. Это стандартный класс Java под названием `java.io.FileOutputStream`, который записывает байтовый поток в файл.

Deserializer. Класс `Deserializer` отвечает за чтение сериализованного байтового потока и создание копии объекта `GameModel` и других объектов, которые были сериализованы.

FileInputStream. Это стандартный класс языка Java под названием `java.io.FileInputStream`, который читает байтовый поток из файла.

Опишем последовательность событий для ситуации, когда пользователь отправляет запрос на сохранение игры в файл или восстановление ее из файла.

1. Игрок сообщает пользовательскому интерфейсу, что он хочет сохранить игру в файле. Тогда объект `UserInterface` создает объект `Serializer`, передавая его конструктору имя файла и ссылку на объект `GameModel`. Объект `Serializer` создает объект `ObjectOutputStream` и объект `FileOutputStream`. `Serializer` использует объект `ObjectOutputStream` для сериализации объекта `GameModel` и всех других связанных с игрой объектов, на которые он ссылается. Для записи этого байтового потока в файл он использует объект `FileOutputStream`.

2. Когда игрок хочет восстановить игру из файла, он сообщает об этом пользовательскому интерфейсу. Тогда объект `UserInterface` создает объект `Deserializer`, передавая его конструктору имя файла и ссылку на объект `GameModel`. Объект `Deserializer` создает объект `ObjectInputStream` и объект `FileInputStream`. Он использует объект `FileInputStream` для чтения сериализованного байтового потока из файла. Для десериализации из байтового потока объекта `GameModel` и всех других связанных с игрой объектов, на которые он ссылается, объект `Deserializer` использует объект `ObjectInputStream`.

Почти все приводимые в этой книге шаблоны описывают только один способ решения проблемы. Шаблон Snapshot отличается от них тем, что предоставляет два способа решения проблемы создания копии состояния объекта.

МОТИВЫ

☺ Нужно сделать моментальный снимок состояния объекта и, кроме того, иметь возможность восстанавливать состояние объекта.

☺ Необходимо, чтобы механизм сохранения и восстановления объектов не зависел от внутренней структуры объекта и поэтому такая внутренняя структура могла бы изменяться, не нуждаясь в изменении механизма сохранения-восстановления.

РЕШЕНИЕ

Рассмотрим два основных подхода к решению проблемы сохранения копии состояния объекта и восстановления его состояния из этой копии. Сначала опишем применение объектов `Memento` для создания временной копии частичного состояния объекта. Затем — использование сериализации для восстановления состояния объекта, и потом сравним эти два способа.

На рис. 8.15 представлена основная организация объектов, использующих объекты `Memento` для сохранения и восстановления состояния объекта.

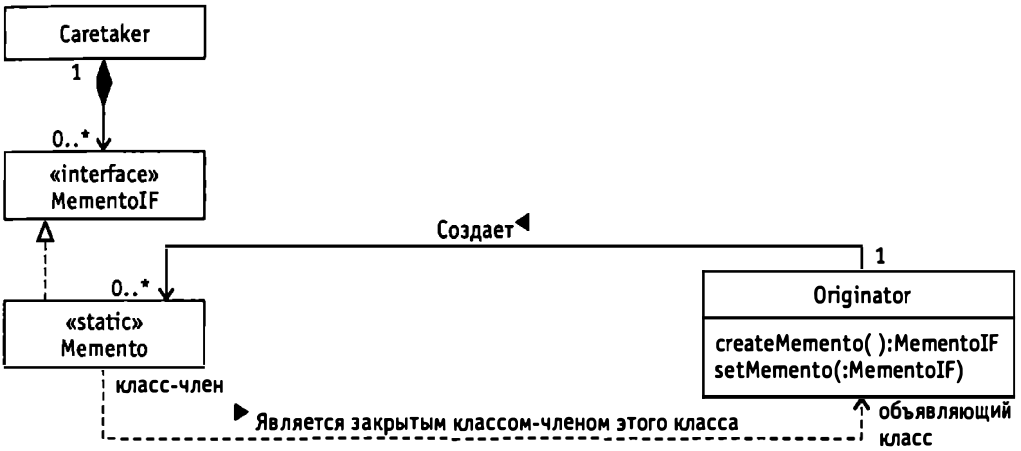


Рис. 8.15. Моментальный снимок, использующий объекты Memento

Опишем роли, исполняемые изображенными на рис. 8.15 классами в рамках некоторой версии шаблона Snapshot, использующей объекты `Memento`.

Originator. Класс в этой роли представляет собой класс, информация о состоянии экземпляров которого подлежит сохранению и восстановлению. При вызове метод `createMemento` создает объект `Memento`, который содержит копию информации о состоянии объекта `Originator`. Позднее можно восстановить состояние объекта `Originator`, передавая объект `Memento` его методу `setMemento`.

Memento. Класс в этой роли представляет собой закрытый статический класс класса `Originator`, реализующий интерфейс `MementoIF`. Его задачей является инкапсуляция моментальных снимков состояния объекта `Originator`. Он является закрытым членом класса `Originator`, поэтому к нему может обращаться только класс `Originator`. Другие классы должны осуществлять доступ к экземплярам класса `Memento` либо как экземпляры класса `Object`, либо через интерфейс `MementoIF`.

MementoIF. Все классы, кроме класса `Originator`, обращаются к объектам `Memento` через этот интерфейс. Интерфейсы в этой роли обычно не объявляют

методов. Если они объявляют какие-либо методы, такие методы не должны допускать изменения инкапсулированного состояния, что гарантирует совместимость и целостность информации о состоянии. Интерфейсы в этой роли проектируются при помощи шаблона Read-Only Interface (см. гл. 6).

Caretaker. Экземпляры классов в роли `Caretaker` поддерживают коллекцию объектов `Memento`. После того как объект `Memento` создан, он обычно добавляется в коллекцию объекта `Caretaker`. Если должна выполняться операция отмены, объект `Caretaker` обычно сотрудничает с другим объектом для выбора объекта `Memento`. После выбора объекта `Memento` именно объект `Caretaker`, как правило, вызывает метод `setMemento` объекта `Originator` для восстановления его состояния.

Другой механизм создания моментального снимка состояния объекта — это сериализация (рис. 8.16). Сериализация отличается от большинства других объектно-ориентированных технологий тем, что при работе она нарушает инкапсуляцию объекта, подлежащего сериализации. Большая (но не обязательно вся) часть нарушения инкапсуляции производится из-за механизма отражения языка Java.

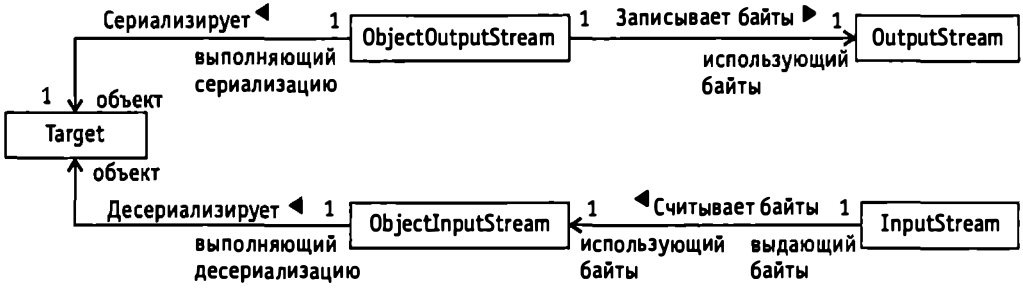


Рис. 8.16. Моментальный снимок с применением сериализации

Опишем роли, исполняемые классами в той версии шаблона Snapshot, которая применяет сериализацию.

Target. Объект `ObjectOutputStream` превращает состояние экземпляров классов, выступающих в этой роли, в байтовый поток. Объект `ObjectInputStream` восстанавливает состояние экземпляров классов, выступающих в этой роли, из байтового потока. Роль объекта `Target` при выполнении таких действий пассивная. Все работу выполняет объект `ObjectOutputStream` или объект `ObjectInputStream`.

ObjectOutputStream. Выступающий в этой роли класс обычно представляет собой стандартный класс языка Java — `java.io.ObjectOutputStream`. Он находит и осуществляет доступ к информации о состоянии объекта `Target`, а также записывает ее в байтовый поток вместе с дополнительной информацией, которая позволяет объекту `ObjectInputStream` восстанавливать информацию о состоянии.

OutputStream. Объект в этой роли является экземпляром подкласса стандартного класса Java — `java.io.OutputStream`. Если информация о состоянии должна сохраняться в течение неопределенного времени, то роль объекта `OutputStream` может выполнять объект `FileOutputStream`. Если информация о состоянии должна сохраняться только во время работы программы, то роль объекта `OutputStream` может исполнять объект `ByteArrayOutputStream`.

ObjectInputStream. Выступающий в этой роли класс представляет собой стандартный класс Java — `java.io.ObjectInputStream` или его подкласс. Экземпляры таких классов читают из байтового потока сериализованную информацию о состоянии и восстанавливают ее.

Если программист не замещает поведение, задаваемое по умолчанию, то объект `ObjectInputStream` помещает исходную информацию о состоянии объекта `Target` в новый экземпляр класса объекта `Target`. При помощи технологий, описанных далее в разделе «Реализация», можно заставить объекты `ObjectInputStream` восстанавливать сохраненное состояние в существующем экземпляре класса `Target`.

В табл. 8.3 представлены некоторые ключевые различия между двумя способами создания и управления моментальными снимками состояния объекта.

Таблица 8.3. Сопоставление двух способов сохранения состояния

Критерии	Сериализация	Использование объектов Memento
Постоянство	Можно использовать для постоянного сохранения состояния, записывая сериализированные данные в файл	Не обеспечивает постоянства
Сложность реализации	Иногда более простой способ при сохранении полного состояния объекта. Это особенно справедливо для объектов, которые содержат ссылки на другие объекты, состояние которых тоже должно быть сохранено	Зачастую более простой способ для записи частичного состояния объекта
Уникальность	Абсолютная уникальность объектов будет потеряна, если не предоставить дополнительный код, препятствующий этому. По умолчанию задается способ, который состоит в том, что сериализация восстанавливает состояние объекта, создавая его копию. Если исходный объект содержит другие объекты и существуют многочисленные ссылки на один и тот же объект, то объект-дубликат будет содержать ссылки на идентичный, но другой объект. Что касается объектов, на которые ссылается восстановленный объект, то здесь сериализация сохраняет относительную уникальность объектов	Уникальность объектов сохраняется. Использование объектов `Memento` — это более простой способ восстановления состояния объекта, потому что он ссылается на тот же объект, на который ссылался до этого

Таблица 8.3. Окончание

Критерии	Сериализация	Использование объектов Memento
Затраты	Значительно увеличивает затраты на создание моментального снимка, сохраняемого в памяти. Большая часть затрат объясняется тем, что сериализация применяет механизм отражения языка Java и при восстановлении состояния создает новые объекты	Не требует особых затрат
Необходимая компетенция	В тех случаях, когда нужно сделать копию полного состояния объекта и всех объектов, участвующих в реализации интерфейса `Serializable`, и сохранение уникальности объекта не имеет большого значения, сериализация требует минимальных знаний. По мере изменения ситуации необходимый уровень компетентности резко возрастает. Некоторые ситуации могут потребовать глубокого знания внутренних характеристик сериализации, механизма отражения и других скрытых аспектов языка Java	Не требует специальных знаний

РЕАЛИЗАЦИЯ

Использование объектов `Memento` для выполнения моментальных снимков состояния объекта реализуется очень просто. Применение сериализации требует большей компетентности и иногда большей сложности. Полное описание сериализации выходит за рамки данной книги. Приведем описание некоторых характеристик сериализации, необходимых при рассмотрении шаблона Snapshot.

Чтобы выполнить сериализацию объекта, нужно создать объект `ObjectOutputStream`, передавая объект `OutputStream` его конструктору. Это делается примерно так:

```
FileOutputStream fout = new FileOutputStream("filename.ser");
ObjectOutputStream obOut = new ObjectOutputStream(fout);
```

Объект `ObjectOutputStream` будет записывать создаваемый им байтовый поток в объект `OutputStream`, переданный его конструктору.

После создания объекта `ObjectOutputStream` можно сериализовать объект, передавая этот объект методу `writeObject` объекта `ObjectOutputStream`, примерно таким образом:

```
ObOut.writeObject(foo);
```

Метод `writeObject` использует механизм отражения языка Java для обнаружения переменных экземпляра объекта, на который ссылается `foo`, и обращается к ним. Если значение переменной экземпляра имеет простой тип, например,

int или double, оно записывается прямо в байтовый поток. Если значение переменной экземпляра — это ссылка на другой объект, то метод writeObject сериализует также и тот объект.

Преобразование сериализованного байтового потока в объект называется десериализацией. Чтобы десериализовать байтовый поток, создается объект ObjectInputStream путем передачи объекта InputStream его конструктору примерно таким образом:

```
FileInputStream fin = new FileInputStream("filename.ser");

ObjectInputStream obIn = new ObjectInputStream(fin);
```

Объект ObjectInputStream будет считывать байты из входного потока, переданного его конструктору.

Если создан объект ObjectInputStream, можно десериализовать связанный с ним байтовый поток, вызывая его метод readObject, например, так:

```
GameModel g = (GameModel)obIn.readObject();
```

Метод readObject возвращает ссылку на Object. Скорее всего, будет нужно, чтобы возвращаемый методом readObject объект был экземпляром какого-то конкретного класса, поэтому придется выполнить операцию приведения типов.

Еще один момент, который необходимо учитывать при сериализации объекта: экземпляр класса можно сериализовать только тогда, когда класс разрешает сериализацию. Класс допускает сериализацию своих экземпляров в том случае, если он реализует интерфейс java.io.Serializable:

```
import java.io.serializable;

...

class foo extends bar implements Serializable {
```

Интерфейс Serializable — это маркер-интерфейс. Он не объявляет никаких членов. Самый простой способ указать, что экземпляры некоторого класса могут быть сериализованы, состоит в том, чтобы объявить этот класс реализующим интерфейс Serializable. Если передать объект, не реализующий интерфейс Serializable, методу writeObject объекта ObjectOutputStream, то этот метод сгенерирует исключение.

До сих пор сериализация выглядела достаточно простой. Существует много ситуаций, для которых вышеописанные детали — это все, что нужно знать, но встречаются и более сложные случаи.

При сериализации объекта задаваемое по умолчанию поведение предполагает также сериализацию всех объектов, на которые он ссылается, и всех объектов, на которые те ссылаются и т.д. до тех пор, пока не будет сериализован весь набор. Несмотря на то что объект может быть экземпляром класса, реализующего интерфейс Serializable, он может ссылаться на какие-либо объекты, не являющиеся Serializable, тогда любая попытка сериализовать объект обречена

на неудачу. Ошибка возникает в том случае, когда метод `writeObject` объекта `ObjectOutputStream` рекурсивно вызывает сам себя с целью сериализации объекта, который не может быть сериализован, и генерирует исключение. Эта проблема решается следующим образом: можно задать механизм сериализации таким образом, чтобы он игнорировал некоторые экземплярные переменные объекта. Проще всего в этом случае объявить переменную с модификатором `transient`, например, так:

```
transient ObjectOutputStream obOut;
```

Механизм сериализации игнорирует переменные с модификатором `transient`, поэтому сериализация проходит успешно, даже если такая переменная ссылается на объект, который не может быть сериализован.

Существует также другая проблема, которая может быть решена посредством объявления некоторых переменных экземпляра как `transient`. Экземпляры ряда классов ссылаются на какие-то объекты, ссылающиеся на многие другие объекты, которые не должны быть сохранены. Сериализация таких объектов просто приводила бы к увеличению затрат на сериализацию и десериализацию. Если значение переменной экземпляра не должно быть частью сериализуемого состояния объекта, то объявление этой переменной экземпляра как `transient` позволяет избежать затрат, связанных с ненужной сериализацией и десериализацией значений этой переменной.

Объявление экземплярных переменных как `transient` способствует решению некоторых проблем, связанных с сериализацией. Но появляется проблема, связанная с десериализацией. Сериализованный байтовый поток не содержит значений для переменных с модификатором `transient`. Если не предпринять мер, после десериализации такие переменные объекта будут содержать значения по умолчанию, соответствующие объявленному типу. Например, переменные числового типа будут иметь значение 0. Если для таких переменных задан объектный тип, их значением будет `null`. Сериализация игнорирует инициализацию и конструкторы.

Класс `ObjectOutputStream` предоставляет механизмы, позволяющие изменять задаваемый по умолчанию способ, в соответствии с которым десериализация управляет переменными с модификатором `transient`. Можно создать код, который будет выполняться уже после того, как сериализация произвела свои действия, задаваемые по умолчанию. Часто при написании такого кода требуется дополнительная информация для восстановления значений с модификатором `transient` переменных, которая не предоставляется при задаваемой по умолчанию сериализации. Класс `ObjectOutputStream` предусматривает механизмы, позволяющие добавлять дополнительную информацию к тем данным, которые обеспечиваются задаваемыми по умолчанию действиями сериализации. Если можно будет добавлять информацию в сериализованный байтовый поток и она будет являться достаточной для восстановления значений переменных объекта с модификатором `transient`, то проблема будет решена.

Чтобы добавить информацию к тем данным, которые обычно предоставляются методом `writeObject` класса `ObjectOutputStream`, можно добавить в сериа-

лизуемый класс метод, который будет вызываться методом `writeObject`. Если объект является экземпляром класса, определяющего соответствующим образом метод `writeObject`, то вместо того, чтобы решать, как управлять экземплярными переменными объекта внутри себя, объект `ObjectOutputStream` вызывает метод `writeObject` этого объекта. Это позволяет классу определять, как будут сериализованы его собственные экземплярные переменные.

Чтобы взять на себя ответственность за сериализацию экземплярных переменных своих экземпляров, класс должен иметь метод следующего вида:

```
private void writeObject(ObjectOutputStream stream)
             throws IOException {
   stream.defaultWriteObject();
...
} // writeObject(ObjectOutputStream)
```

Метод должен быть закрытым, чтобы объект `ObjectOutputStream` мог его распознать. Такие закрытые методы `writeObject` отвечают за запись только тех экземплярных переменных, которые объявлены в своих собственных классах. Эти методы не отвечают за переменные, объявленные в суперклассах.

Почти все закрытые методы `writeObject` прежде всего должны вызвать метод `defaultWriteObject` объекта `ObjectOutputStream`. При обращении к этому методу объект `ObjectOutputStream` должен выполнять свои задаваемые по умолчанию действия сериализации для вызвавшего его класса. Затем закрытый метод `writeObject` вызывает другие методы объекта `ObjectOutputStream` для записи дополнительной информации, необходимой для восстановления значений временных переменных. Класс `ObjectOutputStream` является подклассом класса `DataOutputStream`, поэтому он наследует методы, предназначенные для записи строк и всех простых типов данных.

Чтобы иметь возможность использовать дополнительную информацию во время десериализации, класс должен также определять метод `readObject` примерно так:

```
private void readObject(ObjectInputStream stream)
             throws IOException {
   try {
      stream.defaultReadObject();
   } catch (ClassNotFoundException e) {
      ...
   } // try
   ...
}  // readObject(ObjectInputStream)
```

Метод `writeObject` должен быть закрытым, и поэтому метод `readObject` тоже должен быть закрытым, чтобы его можно было распознать. Он начинает с вызова метода `defaultReadObject` объекта `ObjectInputStream`. При обра-

щении к этому методу объект ObjectInputStream должен выполнять свои задаваемые по умолчанию действия десериализации для вызвавшего его класса. Затем закрытый метод readObject вызывает другие методы объекта ObjectInputStream с целью прочтения некоторой дополнительной информации, которая была предоставлена для восстановления значений переменных. Класс ObjectInputStream является подклассом класса DataInputStream, поэтому он наследует методы, предназначенные для чтения строк и всех простых типов данных.

Приведем пример, демонстрирующий совместную работу этих закрытых методов:

```
public class TextFileReader implements Serializable {
   private transient RandomAccessFile file;
   private String browseFileName;
   ...
   private
   void writeObject(ObjectOutputStream stream)
            throws IOException {
      stream.defaultWriteObject();
      stream.writeLong(file.getFilePointer());
   } // writeObject(ObjectOutputStream)

   private
   void readObject(ObjectInputStream stream)
            throws IOException {
      try {
         stream.defaultReadObject();
      } catch (ClassNotFoundException e) {
         String msg = "Unable to find class";
         if (e.getMessage() != null)
            msg += ": " + e.getMessage();
         throw new IOException(msg);
      } // try
      file = new RandomAccessFile (browseFileName,
            "r");
      file.seek(stream.readLong());
   } // readObject(ObjectInputStream)
} // class TextFileReader
```

Этот класс называется TextFileReader. Он имеет переменную экземпляра file, которая ссылается на объект RandomAccessFile. Класс RandomAccessFile не реализует интерфейс Serializable. Чтобы экземпляры класса TextFileReader могли быть успешно сериализованы и десериализованы, для

закрытым, другим классам запрещено использовать этот конструктор, что заставляет их получать экземпляр класса GameModel посредством вызова его метода getGameModel.

Когда объект ObjectInputStream десериализует байтовый поток, он использует специальный механизм, который создает объекты, не обращаясь к конструкторам и не проверяя значения инициализированных переменных. Если у него есть экземпляр класса, он присваивает экземплярным переменным объекта значения, найденные им в сериализованном байтовом потоке.

После того как объект ObjectInputStream заканчивает десериализацию объекта, он проверяет, содержит ли этот объект метод readResolve. Если объект содержит этот метод, то объект ObjectInputStream вызывает метод readResolve этого объекта и возвращает объект, возвращаемый методом readResolve (а не вновь созданный объект).

Приведем также метод readResolve класса GameModel, который присваивает экземплярным переменным объекта GameModel программы значения, полученные от десериализованного объекта GameModel. Затем он возвращает объект GameModel программы.

```
public Object readResolve() {
      GameModel theModel = getGameModel();
      theModel.mementoManager = mementoManager;
      ...
      return theModel;
   } // readResolve()
```

Остальная часть класса GameModel, связанная с шаблоном Snapshot, участвует в управлении объектами Memento.

В реализации класса MilestoneMemento следует отметить несколько важных моментов. Это закрытый класс класса GameModel. Он запрещает любым другим классам, кроме класса GameModel, получать прямой доступ к своим членам. Класс MilestoneMemento объявляется как статический. Эта незначительная оптимизация позволяет сэкономить затраты на поддержание объектами MilestoneMemento ссылки на содержащий их объект GameModel.

Еще один аспект реализации класса MilestoneMemento состоит в малом количестве методов доступа. Считается хорошим тоном позволять другим классам обращаться к экземплярным переменным объекта только через методы доступа (чтения и записи). Такая практика позволяет получать оптимально инкапсулированные объекты. Из-за тесной связи между классом MilestoneMemento и классом GameModel объекты GameModel имеют прямой доступ к экземплярным переменным объектов MilestoneMemento. Другие классы могут обращаться к классу MilestoneMemento только через интерфейс MilestoneMementoIF, поэтому экземплярные переменные объекта MilestoneMemento скрыты от всех остальных классов.

```
private static class MilestoneMemento
            implements MilestoneMementoIF {
  private String description;
...
  /**
   * Constructor
   * @param description
   *     Причина создания этого объекта.
   */
    MilestoneMemento(String description) {
       this.description = description;
    } // constructor(String)

  /**
   * Возвращает причину создания этого объекта memento.
   */

    public String getDescription() { return description; }
    // Значения этих переменных задаются объектом GameModel.
    MilestoneMementoManager mementoManager;
    ...
  } // class MilestoneMemento
```

Желательно иметь возможность сериализовать объекты `MilestoneMemento`. Несложно заметить, что класс `MilestoneMemento` не реализует интерфейс `Serializable`. В этом нет необходимости, так как он реализует интерфейс `MilestoneMementoIF`, который является расширением интерфейса `Serializable`.

Теперь напишем методы, предоставляемые классом `GameModel` для создания объектов `Memento` и для восстановления состояния из объектов `Memento`:

```
/**
 * Создадим объект Memento, который инкапсулирует
 * состояние этого объекта.
 */
MilestoneMementoIF createMemento(String description) {
  // Создаем объект Memento и задаем значения
  // его экземплярных переменных.
  MilestoneMemento memento;
  memento = new MilestoneMemento(description);
  memento.mementoManager = mementoManager;
```

```
        return memento;
    } // createMemento(String)

    /**
     * Восстановим состояние этого объекта из данного объекта
     * Memento.
     */
    void setMemento(MilestoneMementoIF memento) {
        MilestoneMemento m = (MilestoneMemento)memento;
        mementoManager = m.mementoManager;
    } // setMemento(MilestoneMemento)
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ SNAPSHOT

Command. Шаблон Command позволяет отменять изменения состояния команда за командой, не нуждаясь в выполнении моментального снимка полного состояния объекта после каждой команды.

Read-Only Interface. Интерфейсы, выступающие в роли `MementoIF`, проектируются при помощи шаблона Read-Only Interface.

Observer (Наблюдатель)

Шаблон Observer давно известен и широко используется. Важно знать о шаблоне Observer при работе с уже существующими проектами, использующими этот шаблон.

Шаблон Observer ранее был описан в работе [GoF95].

СИНОПСИС

Позволяет объектам динамически регистрировать зависимости между объектами. В результате объект будет оповещать зависящие от него объекты об изменении своего состояния.

КОНТЕКСТ

Предположим, что создается ПО для компании, которая изготавливает детекторы дыма, сенсоры движения и другие приборы безопасности. Чтобы с выгодой использовать преимущества нового рынка, компания планирует ввести новую линию устройств. Такие устройства смогут посылать сигнал на карту безопасности, которая может быть установлена на большинстве компьютеров. Ожидается, что компании, изготавливающие контролирующие системы безопасности, будут внедрять такие устройства и карты в свои системы. Чтобы облегчить процесс внедрения карт в контролирующие системы, нужно создать удобный пользовательский API.

API должен достаточно легко внедряться в программы будущих заказчиков с тем, чтобы они могли получать извещения от карты безопасности. Работа API не должна требовать от заказчиков изменения архитектуры уже имеющегося у них ПО. API может знать о ПО заказчика только то, что по крайней мере один или, возможно, несколько объектов будут иметь метод, который должен вызываться при получении извещения от устройства безопасности. На рис. 8.17 представлен проект такого API.

Экземпляры класса `SecurityNotifier` получают извещения от карты безопасности и извещают те объекты, которые ранее зарегистрировались для получения извещений. Только объекты, реализующие интерфейс `SecurityObserver`, могут зарегистрироваться у объекта `SecurityNotifier` на получение извещений от этого объекта. Объект `SecurityObserver` считается зарегистрированным на получение извещений от объекта `SecurityNotifier` в том случае, если он передается методу `addObserver` объекта `SecurityNotifier`. При этом отменяется регистрация объекта `SecurityObserver` на получение извещений.

Объект `SecurityNotifier` передает извещение объекту `SecurityObserver`, вызывая его метод `notify`. В качестве параметров он передает методу `notify`

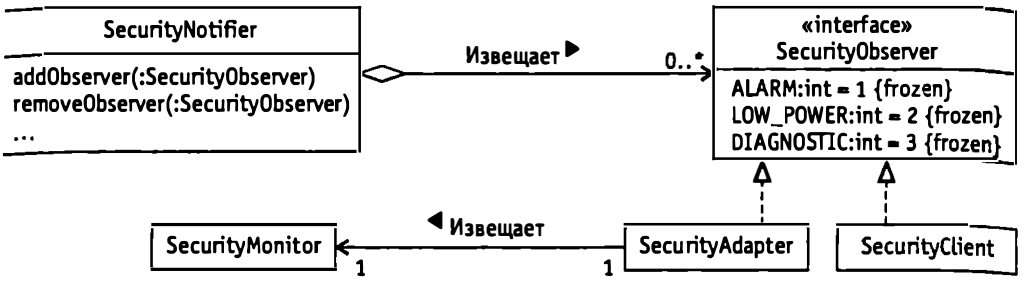


Рис. 8.17. API, предназначенный для извещений системы безопасности

номер, который уникально идентифицирует устройство безопасности, передавшее исходное извещение, и число, задающее тип извещения.

Остальные классы, изображенные на диаграмме, не имеют отношения к API. Это классы, которые уже существуют или должны быть добавлены к предполагаемому контролирующему ПО заказчика. Класс, показанный на диаграмме как `SecurityClient`, может обозначать любой класс, который заказчик добавляет к своему контролирующему ПО и который реализует интерфейс `SecurityObserver`. Потребители могут добавлять такие классы к своему контролирующему ПО для обработки извещений, поступивших от объекта `SecurityNotifier`.

Класс, указанный на диаграмме как `SecurityMonitor`, соответствует любому классу, имеющемуся в контролирующем ПО заказчика и не реализующему интерфейс `SecurityObserver`, но содержащему метод, который должен вызываться для обработки извещений от устройств безопасности. Потребитель может заставить экземпляры такого класса получать извещения, не изменяя сам класс. С этой целью потребитель может написать класс-адаптер, реализующий интерфейс `SecurityObserver` таким образом, что его метод `notify` будет вызывать соответствующий метод класса `SecurityMonitor`.

МОТИВЫ

☺ Реализуются два разных независимых класса. Экземпляр одного такого класса должен иметь возможность извещать другие объекты об изменении своего состояния. Экземпляр другого класса должен быть оповещен, когда связанный с ним объект изменяет состояние. Однако эти два класса не должны работать друг с другом. Чтобы быть многократно используемыми, им не следует иметь никакой непосредственной информации друг о друге.

☺ Есть отношение зависимости «один к нескольким», которое может потребовать от объекта оповещения некоторых зависящих от него объектов об изменении его состояния.

☹ Для передачи извещений и задания их приоритета нужна некоторая логика. Эта логика не зависит ни от отправителя, ни от получателя извещений.

РЕШЕНИЕ

На рис. 8.18 представлена диаграмма классов шаблона Observer.

Очевидно, что эта диаграмма более сложна, чем изображенная на рис. 8.17, которая включает некоторые упрощения, описанные в разделе «Реализация».

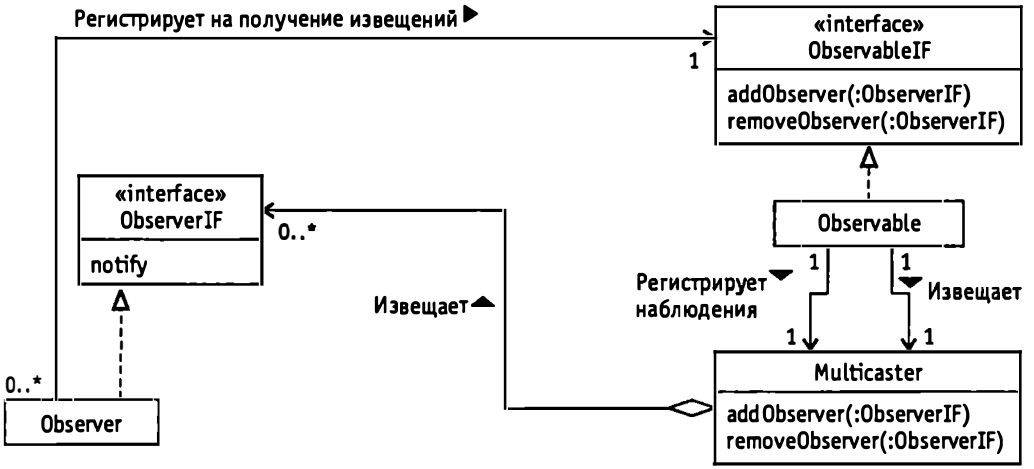


Рис. 8.18. Шаблон Observer

Опишем роли, исполняемые классами и интерфейсами в шаблоне Observer.

ObserverIF. Интерфейс в этой роли определяет метод, который обычно называется `notify` или `update`. Объект `Observable` вызывает этот метод для отправки извещения об изменении его состояния, передавая ему любые необходимые аргументы. Во многих случаях ссылка на объект `Observable` — это один из аргументов, позволяющий методу знать, какой объект отправил извещение.

Observer. Экземпляры классов в этой роли реализуют интерфейс `ObserverIF` и получают извещения об изменении состояния от объектов `Observable`.

ObservableIF. Объекты `Observable` реализуют интерфейс, выступающий в этой роли. Интерфейс определяет два метода, позволяющие объектам `Observer` регистрироваться или отменять регистрацию на получение извещений.

Observable. Класс в этой роли реализует интерфейс `ObservableIF`. Его экземпляры отвечают за управление регистрацией объектов `ObserverIF`, которые хотят получать извещения об изменении состояния. Кроме того, его экземпляры отвечают за передачу этих извещений. Класс `Observable` непрямым образом реализует эти обязанности. Вместо этого он делегирует вышеуказанные обязанности объекту `Multicaster`.

Multicaster. Экземпляры класса в этой роли управляют регистрацией объектов `ObserverIF` и передачей им извещений от имени объекта `Observable`. Суть

этой роли в том, чтобы повысить возможности многократного использования кода. Делегирование этих обязанностей классу `Multicaster` позволяет многократно использовать их реализации всем классам `Observable`, реализующим один и тот же интерфейс `ObservableIF`, или передавать извещения объектам, реализующим один и тот же интерфейс `ObserverIF`.

На рис. 8.19 представлены все случаи взаимодействия между объектами, участвующими в шаблоне Observer.

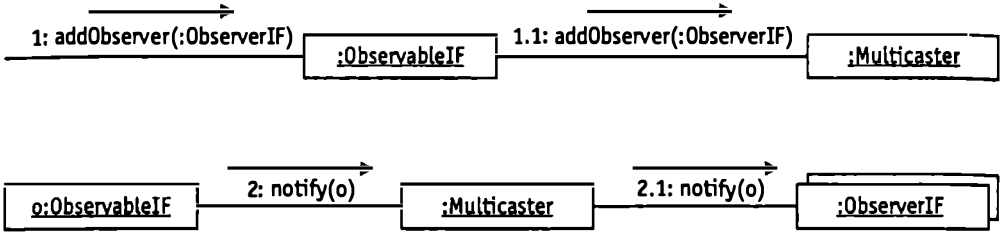


Рис. 8.19. Взаимодействие в шаблоне Observer

Опишем эти взаимодействия.

1. Объекты, реализующие интерфейс `ObserverIF`, передаются методу `addObserver` объекта `ObservableIF`.

   1.1. Объект `ObservableIF` делегирует вызов метода `addObserver` связанному с ним объекту `Multicaster`. Объект `Multicaster` добавляет объект `ObservableIF` в поддерживаемую им коллекцию объектов `ObserverIF`.

2. Объект `ObservableIF`, помеченный буквой `o`, должен оповещать другие объекты об изменении своего состояния. Передачу извещения он инициирует посредством вызова метода `notify` связанного с ним объекта `Multicaster`.

   2.1. Объект `Multicaster` вызывает метод `notify` каждого из объектов `ObserverIF`, принадлежащих его коллекции.

РЕАЛИЗАЦИЯ

Наблюдение за объектом Observable

Объект `Observable` в качестве параметра метода `notify` объекта `Observer` обычно передает ссылку на самого себя. В большинстве случаев объект `Observer` должен обращаться к атрибутам объекта `Observable` с целью реагирования на извещения. Опишем несколько способов предоставления такого доступа.

- Добавление методов в интерфейс `ObservableIF` для считывания значений атрибутов. Как правило, это самое лучшее решение. Однако оно работает

только в том случае, если все классы, реализующие интерфейс ObservableIF, имеют общий набор атрибутов, достаточный для того, чтобы объекты Observer могли воздействовать на извещения.

- Может быть несколько интерфейсов ObservableIF, каждый из которых будет обеспечивать доступ к такому количеству атрибутов, которое является достаточным для воздействия объектов Observer на извещения. С этой целью интерфейсы ObserverIF должны объявлять некоторую версию своего метода notify для каждого из интерфейсов ObservableIF. Однако при требовании от объектов наблюдателя осведомленности о множестве интерфейсов существование интерфейса ObservableIF во многом теряет свой изначальный смысл. Требовать от класса осведомленности о множестве интерфейсов и о множестве классов — почти одно и то же, поэтому такое решение проблемы — не самый лучший вариант.
- Можно передать атрибуты, необходимые объектам ObserverIF, в качестве параметров их методам notify. Основным недостатком данного решения является то, что в этом случае объекты Observable должны иметь достаточную информацию об объектах ObserverIF, позволяющую предоставлять им правильные значения атрибутов. Если набор атрибутов, необходимых объектам ObserverIF, изменяется, то должны соответствующим образом изменяться все классы Observable.
- Существует возможность вообще обойтись без интерфейса ObservableIF и передать объекты Observable объектам ObserverIF в качестве экземпляров их реального класса. Это подразумевает перезагрузку метода notify интерфейса ObserverIF таким образом, что для каждого класса Observable, передающего извещения объектам ObserverIF, определяется свой метод notify.

Основной недостаток такого подхода состоит в том, что классы Observer должны быть осведомлены о классах Observable, которые будут передавать извещения их экземплярам, и должны знать, как считывать из них необходимые им атрибуты. Но с другой стороны, если только один класс Observable будет передавать извещения классам Observer, то такое решение будет являться самым лучшим. Оно не приведет к усложнению каких-либо классов. Зависимость от единственного интерфейса заменяется зависимостью от единственного класса. Проект становится проще благодаря устранению интерфейса ObservableIF.

В примере, описанном в разделе «Контекст», используется такое упрощенное решение.

Отказ от класса Multicaster

Другим распространенным упрощением шаблона Observer является устранение класса Multicaster. Если класс Observable представляет собой единственный класс, передающий извещения объектам, реализующим определенный интерфейс, то нет никакой необходимости в классе Multicaster. Именно поэтому

пример, описанный в разделе «Контекст», не содержит класс, выступающий в роли Multicaster. Другая причина отказа от класса Multicaster состоит в том, что объект Observable всегда должен передавать извещения только одному объекту. В таком случае управление и передача извещений объектам Observer настолько проста, что наличие класса Multicaster скорее усложняет, чем упрощает проект.

Пакетирование извещений

Иногда нет необходимости или пользы в том, чтобы извещать объекты Observer о каждом изменении объекта Observable. В таком случае можно избежать передачи ненужных извещений, создавая пакет изменений состояния и откладывая отправку извещений до тех пор, пока не заполнится весь этот пакет изменений состояния. Если состояние объекта Observable изменяется другим объектом, то представление единственного извещения в виде пакета изменений может вызвать затруднения. Нужно добавить в класс объекта Observable метод, который может быть вызван другими объектами для указания начала пакета изменения состояния. Если изменение состояния является только частью пакета, то объект не должен передавать какие-либо извещения своим зарегистрированным наблюдателям. Кроме того, нужно добавить в класс объекта Observable метод, который может быть вызван другими объектами для указания конца пакета изменений состояния. Если при вызове этого метода с момента начала пакета происходили какие-либо изменения состояния, объект должен передать извещения своим зарегистрированным наблюдателям.

Если изменение состояния объекта Observable инициируется несколькими объектами, то определение конца пакета изменений может быть более сложным. Эффективный способ решения данной проблемы заключается в том, чтобы ввести дополнительный объект, который должен координировать изменения состояния, инициируемые другими объектами, и достаточно хорошо понимать их логику с тем, чтобы определять конец пакета изменений. Более подробное описание использования одного объекта для координирования действий других объектов можно найти в разделе, посвященном шаблону Mediator.

Запрет

Обычно шаблон Observer используется для оповещения других объектов об изменении состояния некоторого объекта. Широко распространенная версия этого шаблона состоит в том, что определяется альтернативный интерфейс ObservableIF, который позволяет объектам выдавать запрос на получение извещения, перед тем как изменяется состояние объекта. Как правило, если извещения об изменении состояния отправляются после изменения состояния, это делается с той целью, чтобы разрешить передачу изменения другим объектам. Если извещение отправляется перед изменением состояния, то другие объекты могут запретить изменение состояния. Чаще всего при реализации такого запрета объект должен генерировать исключение, что и препятствует предполагаемому изменению состояния объекта.

СЛЕДСТВИЯ

Шаблон Observer позволяет объекту передавать извещения другим объектам таким образом, что ни отправитель, ни получатель извещений ничего не знают о классах друг друга.

Существуют некоторые ситуации, когда использование шаблона Observer может приводить к непредвиденным и нежелательным результатам.

- Если объект должен передать извещения большому количеству объектов, передача извещений может потребовать много времени. Это объясняется тем, что один объект может иметь множество наблюдателей, непосредственно зарегистрировавшихся на получение его извещений. Это может также произойти, когда какой-то объект имеет множество косвенных наблюдателей и его извещения каскадно передаются другими объектами. Иногда можно уменьшить негативные последствия такой ситуации, делая передачу извещений асинхронной, выполняющейся в своем собственном потоке. Однако асинхронная передача извещений способна породить свои собственные проблемы.
- Более серьезная проблема связана с циклическими зависимостями. Объекты вызывают методы `notify` друг друга до тех пор, пока не переполнится стек и не будет сгенерирована ошибка `StackOverflowError`. Несмотря на всю серьезность этой проблемы, она может быть легко решена посредством задания внутреннего флага в одном из классов, участвующих в цикле. Такой флаг обозначает рекурсивное извещение, например:

```
...
   private boolean inNotify = false;
   public void notify(ObservableIF source) {
      if (inNotify)
         return;
      inMotify = true;
      ...
      inNotify = false;
   }
```

- Если извещение может быть передано асинхронно по отношению к другим потокам (как в случае, описанном в примере раздела «Контекст»), то следует рассмотреть некоторые дополнительные последствия. Механизм асинхронной передачи извещений должен обеспечивать согласованность объектов, получающих извещения. Важно также, чтобы извещение не блокировало другой ожидающий поток.

Когда объект `Observer` получает извещение, он знает, какой объект изменился, но он не знает, каким образом он изменился. Объект `Observer` не должен определять, какие атрибуты объекта `ObservableIF` изменились. Наблюдателю,

как правило, проще взаимодействовать со всеми атрибутами объекта ObservableIF, чем брать на себя ответственность за определение, какие атрибуты изменились, и потом работать только с измененными.

ПРИМЕНЕНИЕ В JAVA API

Модель делегирования событий в языке Java — это специальная форма шаблона Observer. Классы, экземпляры которых могут быть источниками событий, выступают в роли Observable. Интерфейсы получателей событий исполняют роль ObserverIF. Классы, реализующие интерфейсы получателей событий, играют роль Observer. Поскольку существует целый ряд классов, передающих различные подклассы класса java.awt.AwtEvent своим получателям, то существует используемый ими класс в роли Multicaster, имя которого — java.awt.AWTEventMulticaster.

ПРИМЕР КОДА

Следующий код реализует некоторую часть проекта контроля безопасности, представленного в разделе «Контекст». Первый фрагмент кода — это интерфейс SecurityObserver. Чтобы экземпляры некоторого класса могли получать извещения, он должен реализовывать интерфейс SecurityObserver.

```
public interface SecurityObserver {
   public final int ALARM = 1;
   public final int LOW_POWER = 2;
   public final int DIAGNOSTIC = 3;

   /**
    * Этот метод вызывается для передачи извещения
    * от устройства безопасности.
    * @param device
    *     Идентификтор устройства, от которого получено это
    *     извещение.
    * @param event
    *     Одна из вышеупомянутых констант.
    */
   public void notify(int device, int event);
} // interface SecurityObserver
```

Следующий фрагмент кода представляет собой класс SecurityNotifier, отвечающий за передачу извещений, которые компьютер получает от устройств системы безопасности.

```
class SecurityNotifier {
```

```
...
   public void addObserver(SecurityObserver observer) {
      observers.add(observer);
   } // addObserver(SecurityObserver)

   public void removeObserver(SecurityObserver observer) {
      observers.remove(observer);
   } // removeObserver(SecurityObserver)

   private void notify(int device, int event) {
      Iterator iterator = observers.iterator();
      while (iterator. hasNext()) {
         ((SecurityObserver)iterator.next()).notify(device,
               event);
      } // while
   } // notify(int, int)
} // class SecurityNotifier
```

И наконец, пример класса-адаптера, позволяющий экземплярам класса SecurityMonitor получать извещения даже в том случае, когда класс SecurityMonitor не реализует класс SecurityObserver.

```
class SecurityAdapter implements SecurityObserver {
   private SecurityMonitor sm;

   SecurityAdapter(SecurityMonitor sm) {
      this.sm = sm;
   } // Constructor(SecurityMonitor)

   /**
    * Этот метод вызывается для передачи извещения,
    * связанного с безопасностью.
    * @param device
    *     Идентификатор устройства, от которого получено это
    *     извещение.
    * @param event
    *     Одна из вышеупомянутых констант.
    */
   public void notify(int device, int event) {
      switch (event) {
         case ALARM:
            sm.securityAlert(device);
```

```
            break;
         case LOW_POWER:
         case DIAGNOSTIC:
            sm.diagnosticAlert(device);
            break;
      } // switch
   } // notify(int, int)
} // class SecurityAdapter
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ OBSERVER

Adapter. Шаблон Adapter может использоваться для того, чтобы разрешить объектам, не реализующим необходимый интерфейс, участвовать в шаблоне Observer и получать извещения от своего имени.

Delegation. Шаблон Observer использует шаблон Delegation.

Mediator. Шаблон Mediator иногда используется для координации изменений состояния объекта `Observable`, инициируемых несколькими объектами.

Publish-Subscriber. Шаблон Publish-Subscriber (описанный в книге [Grand2001]) — это специальная версия шаблона Observer, предназначенная для передачи извещений удаленным и распределенным объектам.

State (Состояние)

Этот шаблон был описан в работе [GoF95].

СИНОПСИС

Инкапсулирует состояния объекта в виде отдельных объектов, каждый из которых расширяет общий суперкласс.

КОНТЕКСТ

Многие объекты должны иметь динамически изменяющийся набор атрибутов, который называется их *состоянием*. Такие объекты называются *объектами, имеющими состояние*. Как правило, состояние объекта представляет собой один из предварительно заданных наборов значений. Когда имеющий состояние объект узнает о внешнем событии, его состояние может измениться. Поведение объекта, имеющего состояние, в какой-то мере определяется его состоянием.

В качестве примера объекта, имеющего состояние, рассмотрим случай написания диалогового окна для редактирования параметров программы. Окно будет иметь кнопки для внесения выполненных изменений:

- OK, которая сохраняет значения параметров диалогового окна как в файле, так и в рабочих переменных программы;
- Save, которая сохраняет значения параметров только в файле;
- Apply, которая сохраняет значения параметров только в рабочих переменных программы;
- Revert, которая восстанавливает переменные диалогового окна, записывая в них значения из файла.

Можно спроектировать диалоговое окно так, чтобы оно не отслеживало состояний. Если диалоговое окно не отслеживает состояния, то оно всегда будет вести себя одинаково. Кнопка OK будет доступна независимо от того, редактировались или нет значения переменных. Кнопка Revert будет доступна даже в том случае, если пользователь только что изменил значения переменных на те, которые хранились в файле. Если нет других условий, проектирование такого диалогового окна без учета состояний считается удовлетворительным.

В некоторых случаях поведение диалогового окна, не отслеживающего свои состояния, может приводить к появлению проблем. Обновление значений рабочих переменных программы может быть произведено неправильными данными. Сохранение значений параметров в файле может потребовать слишком много

избежать выполнения ненужных операций сохранения в файл или задания неверных значений рабочих параметров программы, нужно сделать диалоговое окно с поддержкой своих состояний. Тогда операции будут разрешены только в том случае, когда файл или рабочие переменные изменяются на данные, отличные от уже хранящихся. На рис. 8.20 показана диаграмма состояний, демонстрирующая четыре состояния, которые необходимы для задания такого поведения.

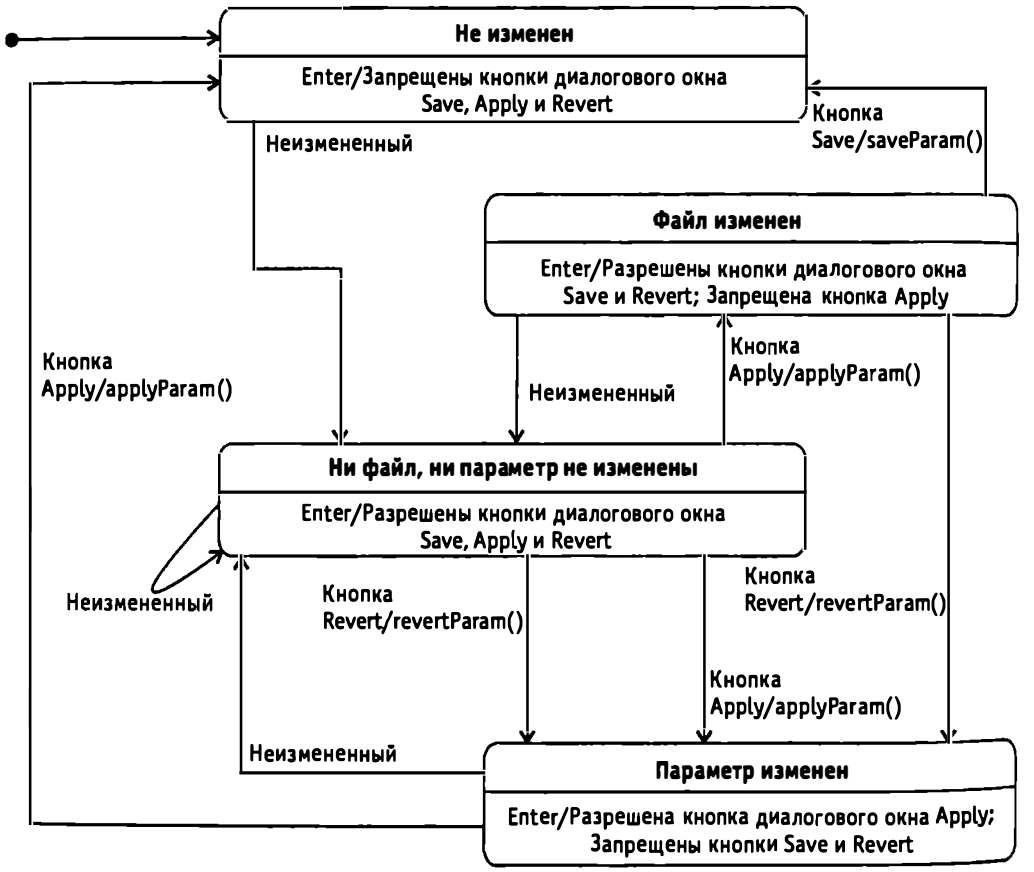


Рис. 8.20. Диаграмма состояний диалогового окна, содержащего параметры

Чтобы реализовать диаграмму состояний, представленную на рис. 8.20, можно спроектировать классы, показанные на рис. 8.21.

На диаграмме классов изображены четыре класса, которые соответствуют четырем состояниям, представленным на диаграмме состояний, и их общий суперкласс. Суперкласс `DirtyState` имеет открытый метод под названием `processEvent`. Метод `processEvent` принимает в качестве аргумента идентификатор события

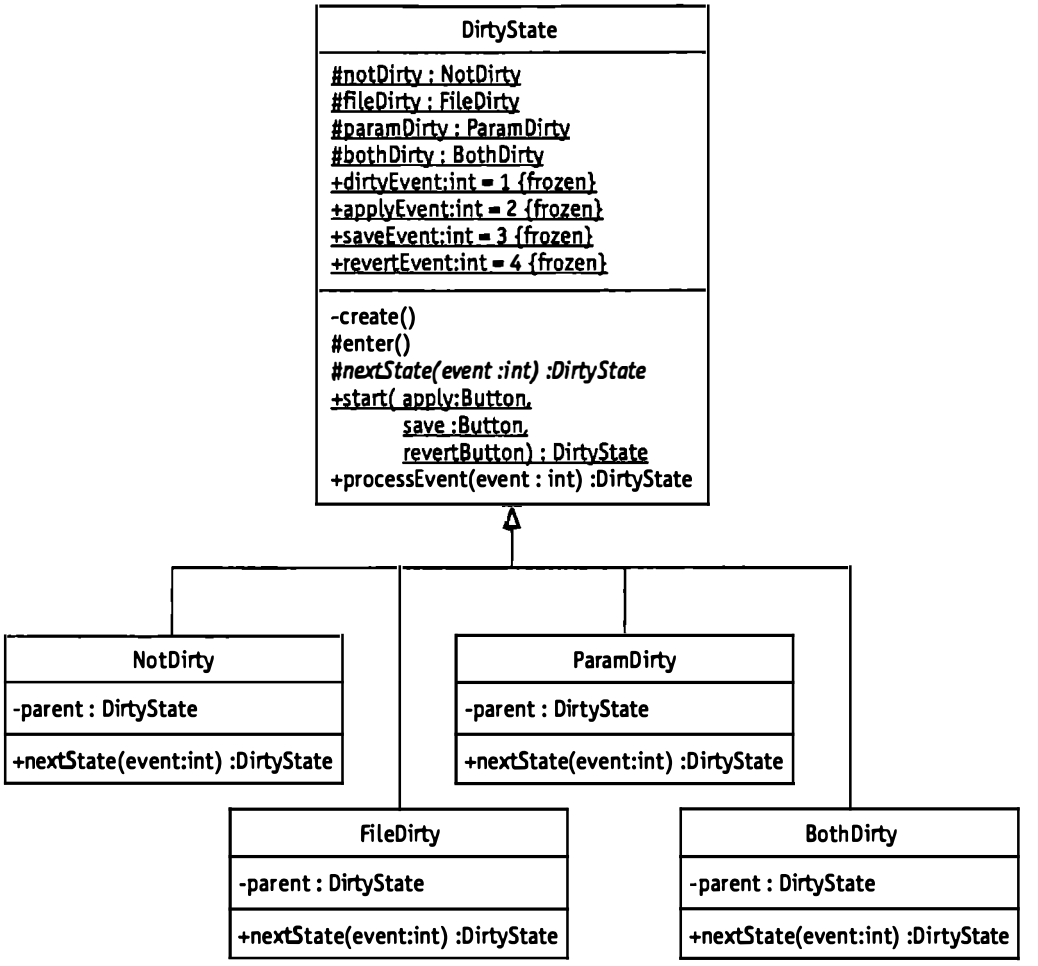


Рис. 8.21. Класс DirtyState

он вызывает абстрактный метод `nextState`. Каждый подкласс класса `DirtyState` соответствующим образом замещает метод `processEvent` для определения следующего состояния. Класс `DirtyState` имеет также статический метод под названием `start`.

Метод `start` активизирует процесс, создавая экземпляр каждого подкласса класса `DirtyState` и возвращая исходное состояние. Кроме того, метод `start` создает экземпляр класса `DirtyState` и инициализирует его переменные `notDirty`, `fileDirty`, `paramDirty` и `bothDirty` соответствующими экземплярами подклассов, которые он создает.

Класс `DirtyState` определяет защищенный метод под названием `enter`. Метод `enter` объекта `DirtyState` вызывается в том случае, когда объект стано-

rocessEvent. Метод enter, задаваемый классом DirtyState, ничего не делает. Однако подклассы замещают метод enter с целью реализации своих действий по инициализации.

Класс DirtyState определяет некоторые статические константы. Константы идентифицируют коды событий, которые передаются методу processEvent.

МОТИВЫ

- Поведение объекта определяется внутренним состоянием, которое изменяется в ответ на события.
- Организация логики, управляющей состоянием объекта, распределяется между подклассами DirtyState, что позволяет избежать появления большого плохо управляемого куска кода.

РЕШЕНИЕ

На диаграмме, представленной на рис. 8.22, изображена основная организация классов для шаблона State.

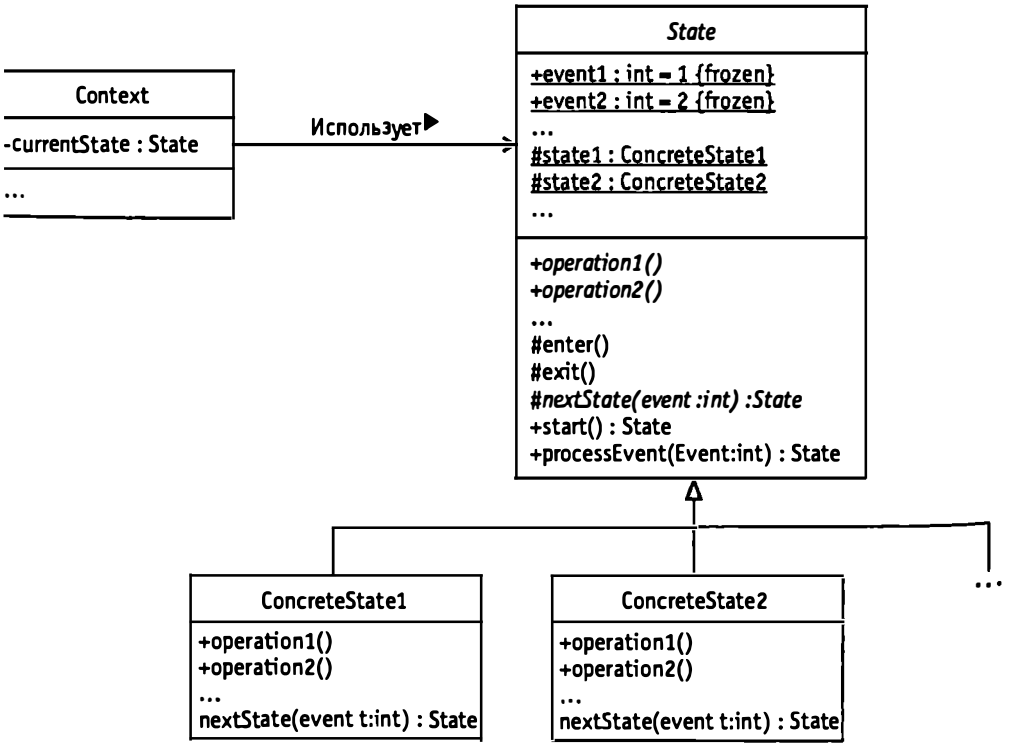


Рис. 8.22. Классы шаблона State

Опишем роли, исполняемые этими классами.

Context. Экземпляры классов в этой роли реализуют поведение, специфичное для конкретного состояния. Экземпляры класса `Context` определяют свое текущее состояние, поддерживая ссылку на экземпляр конкретного подкласса класса `State`. Подкласс класса `State` задает состояние.

State. Класс `State` представляет собой суперкласс для всех классов, используемых для представления состояния объектов `Context`. Класс `State` определяет следующие методы.

- Метод `enter` объекта `State` вызывается в том случае, когда состояние, представленное этим объектом, становится текущим. Класс `State` обеспечивает по умолчанию бездействующую реализацию этого метода. Обычно подклассы класса `State` замещают этот метод.
- Метод `start` выполняет всю необходимую инициализацию объектов управления состоянием и возвращает объект, соответствующий исходному состоянию клиентского объекта. Перед тем как он возвращает объект `State`, представляющий исходное состояние, он вызывает метод `enter` объекта `State`.
- Метод `nextState` — это абстрактный метод, которому передается параметр, описывающий случившееся событие, и который возвращает следующее состояние. Каждый конкретный подкласс класса `State` замещает метод `nextState` для определения нужного следующего состояния.
- Метод `exit` объекта `State` вызывается в том случае, когда состояние, задаваемое этим объектом, перестает быть текущим. Класс `State` обеспечивает реализацию по умолчанию этого метода, которая ничего не делает. Обычно подклассы класса `State` замещают этот метод.
- Метод `processEvent` является открытым методом, который принимает аргумент, описывающий случившееся событие, и возвращает новое текущее состояние. Метод `processEvent` вызывает метод `nextState`. Если объект, возвращаемый методом `nextState`, не является текущим объектом `State`, то текущим станет новое состояние. В этом случае метод `processEvent` вызывает метод `exit` прежнего текущего состояния, а затем вызывает метод `enter` нового текущего состояния.
- Методы `operation1`, `operation2` и т.д. реализуют операции, которые ведут себя по-разному для разных состояний. Например, если объект имеет состояния, связанные с ним и имеющие имена `On` и `Off`, реализация операции для состояния `On` может предусматривать какие-то действия, а реализация для состояния `Off` может предполагать отсутствие действий. Проектирование этих методов относится к сфере деятельности шаблона Polymorphism, описанного в книге [Grand99].

Класс `State` определяет константы, которые являются символьными именами кодов событий, передаваемых методу `processEvent`.

Если класс `State` не имеет экземплярных переменных, нет необходимости иметь более одного экземпляра этого класса. Если существует только один

экземпляр конкретного подкласса класса `State`, то класс `State` будет содержать статическую переменную, которая ссылается на этот экземпляр. Реализации метода `processEvent` не создают новые экземпляры, а возвращают те экземпляры, на которые ссылаются эти переменные.

ConcreteState1, ConcreteState2 и т.д. Это конкретные подклассы класса `State`. Они должны соответствующим образом реализовывать методы `operation1`, `operation2` и т.д.

Кроме того, они должны реализовывать метод `nextState` для определения соответствующего следующего состояния для каждого события. Классы могут замещать метод `enter` и/или метод `exit` с целью реализации соответствующих действий, выполняемых при входе или выходе из состояния.

РЕАЛИЗАЦИЯ

Ни один класс, за исключением класса `State`, не должен знать о подклассах класса `State`. С этой целью подклассы класса `State` объявляются как закрытые классы-члены класса `ContextState`.

СЛЕДСТВИЯ

☺ Код для каждого состояния находится в его собственном классе. Такая организация позволяет легко добавлять новые состояния без непредвиденных последствий. Поэтому шаблон State хорошо работает как для немногочисленных, так и для многочисленных состояний.

☺ Для клиентов объектов состояний переходы состояний должны быть элементарными. Клиент вызывает метод `processEvent` текущего состояния, и тот возвращает клиенту новое состояние.

☺ Процедурная реализация поведения, содержащего состояния, обычно предусматривает наличие нескольких методов, содержащих команды `switch` или цепочки команд `if-else` для распределения кода, зависящего от состояний. Такие цепочки команд иногда могут быть очень длинными и сложными для понимания. Применение шаблона `State` позволяет отказаться от таких команд `switch` и цепочек команд `if-else`. В этом случае логика является более сцепленной, и классы в результате должны иметь меньшие методы.

☺ Применение шаблона State предполагает меньше строк кода и меньше ветвей выполнения. Этот шаблон позволяет упростить тестирование программы при помощи шаблона White Box Testing, описанного в книге [Grand99].

☺ Объекты состояний, которые представляют непараметрические состояния, могут использоваться как одиночки, если нет необходимости в создании нового экземпляра класса `State`. В некоторых случаях (подобных примеру, описанному в разделе «Контекст») нет необходимости создавать экземпляр класса `State` с целью предоставления набору объектов состояний способа

совместного использования данных. Даже в таких случаях для каждого подкласса класса `State`, представляющего непараметрическое состояние, может существовать единственный экземпляр этого класса, связанный с экземпляром класса `State`.

- Использование шаблона State позволяет отказаться от кода в методе, который предназначен для распределения кода, зависящего от состояний. Он не устраняет зависящие от состояний команды `switch`, которые передают управление обработке состояния события.

ПРИМЕР КОДА

Приведем код, который реализует диаграмму классов, описанную в разделе «Контекст» (рис. 8.21).

```
class DirtyState {
   // Символьные константы для событий.
   public static final int DIRTY_EVENT  = 1;
   public static final int APPLY_EVENT  = 2;
   public static final int SAVE_EVENT   = 3;
   public static final int REVERT_EVENT = 4;

   // Символьные константы для состояний.
   private static BothDirty bothDirty;
   private static FileDirty fileDirty;
   private static ParamDirty paramDirty;
   private static NotDirty notDirty;

   private Parameters parameters;
   private Button apply, save, revert;

   /*
    * Этот закрытый конструктор запрещает другим классам
    * (находящимся вне этого класса) создавать экземпляры
    * данного класса.
    */
   DirtyState() {
      if (bothDirty==null) {
         bothDirty = new BothDirty();
         fileDirty = new FileDirty();
         paramDirty = new ParamDirty();
         notDirty = new NotDirty();
      } // if
   } // constructor ()
```

Метод start класса DirtyState инициализирует конечный автомат. Его аргументами являются: объект Parameters, который может использоваться конечным автоматом для обновления рабочих переменных программы, и кнопки, разрешаемые или запрещаемые конечным автоматом. Метод start возвращает исходное состояние.

```
public static DirtyState start(Parameters p,
                               Button apply,
                               Button save,
                               Button revert){
  DirtyState d = new DirtyState();
  d.parameters = p;
  d.apply = apply;
  d.save = save;
  d.revert = revert;
  d.notDirty.enter();
  return d.notDirty;
} // start(Button, Button, Button)

/**
 * Отвечает на данное событие следующим состоянием.
 * @param event Код события.
 * @return Возвращает следующее состояние.
protected DirtyState nextState(int event) {
  // Этот незамещенный метод не должен вызываться никогда.
  throw new IllegalAccessError();
} // nextState(int)

/**
 * Отвечает на данное событие, определяя следующее текущее
 * состояние и переходя к нему, если оно оказывается
 * отличным от текущего.
 */
public final DirtyState processEvent(int event) {
  DirtyState myNextState = nextState(event);
  if (this!=myNextState) {
    myNextState.enter();
  } // if
  return myNextState;
} // processEvent(int)

/**
 * Если данный объект становится текущим состоянием,
```

```
     * вызывается этот метод.
     */
    protected void enter() { }
```

Четыре конкретных подкласса класса `DirtyState` реализуются как закрытые классы. Для краткости здесь приводится только один из этих классов.

```
    /**
     * Класс представляет такое состояние, когда содержимое
     * полей диалога не совпадает с содержимым файла
     * или значениями рабочих параметров.
     */
    private class BothDirty extends DirtyState {
      /**
        * Отвечает на данное событие.
        * @return Возвращает следующее состояние.
      */
      public DirtyState nextState(int event) {
        switch (event) {
          case DIRTY_EVENT:
            return this;
          case APPLY_EVENT:
            if (parameters.applyParam()) {
            fileDirty.enter();
            return fileDirty;
          } // if
        case SAVE_EVENT:
          if (parameters.saveParam()) {
              paramDirty.enter();
              return paramDirty;
          } // if
        case REVERT_EVENT:
          if (parameters.revertParam()) {
              paramDirty.enter();
              return paramDirty;
          } // if
        default:
          String msg = "unexpected event "+event;
          throw new IllegalArgumentException(msg);
      } // switch (event)
    } // nextState(int)
```

```
        /**
         * Если данный объект становится текущим состоянием,
         * вызывается этот метод.
         */
          protected void enter() {
             apply.setEnabled(true);
             revert.setEnabled(true);
             save.setEnabled(true);
          } // enter
        } // class BothDirty
    } // class DirtyState
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ STATE

Flyweight. Шаблон Flyweight можно применять для совместного использования объектов состояний.

Mediator. При реализации пользовательских интерфейсов шаблон State часто применяется вместе с шаблоном Mediator.

Singleton. Непараметрические состояния можно реализовывать с помощью шаблона Singleton.

Polymorphism. Проект операций, зависящих от состояний, реализован с использованием конкретных классов состояний, соответствующих шаблону Polymorphism, рассмотренному в книге [Grand99].

Null Object (Нулевой объект)

Этот шаблон ранее был описан в работе [Woolf97].

СИНОПСИС

Шаблон Null Object представляет собой альтернативу использованию `null` для указания отсутствия объекта, которому делегируется операция. Применение `null` для указания отсутствия такого объекта требует проверки на `null` перед каждым вызовом методов объекта. Вместо использования `null` шаблон Null Object вводит ссылку на объект, который ничего не делает.

КОНТЕКСТ

Допустим, нужно написать некоторые классы, инкапсулирующие бизнес-правила предприятия[1]. Эти классы будут использоваться в различных средах, поэтому необходимо, чтобы такие объекты смогли передавать предупреждения диалоговым окнам, журналу событий, по другим адресам или вообще никуда. Организовать это можно просто: определить интерфейс под названием `WarningRouter`, а затем написать классы, которые делегируют передачу предупреждений объектам, реализующим этот интерфейс (рис. 8.23).

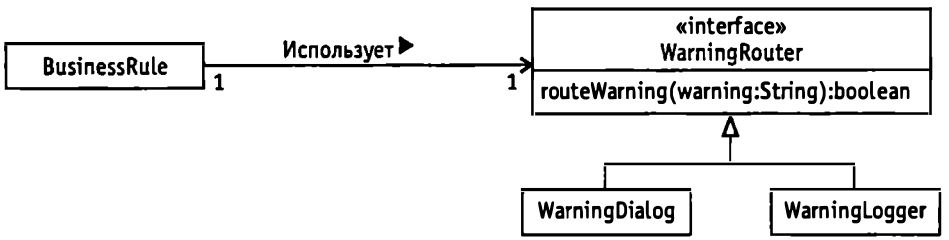


Рис. 8.23. Интерфейс WarningRouter

Чтобы управлять такой ситуацией, когда предупреждающие сообщения не должны никуда не передаваться, можно было бы задать переменную, которая ссылалась бы на объект `WarningRouter`, содержащий `null`. Применение такого

[1] Бизнес-правило — это правило, управляющее поведением информационных систем предприятия. В виде правил могут определяться такие вещи, как время нового заказа товара или рассмотрения платежеспособности клиента. Со временем бизнес-правила меняются, поэтому реализация бизнес-правил должна быть как можно более приспособлена к их изменению.

механизма означает, что перед тем, как объект `BusinessRule` сможет выдать предупреждающее сообщение, он сначала должен проверить, не равна ли переменная `null`. В зависимости от конкретного класса бизнес-правила только одно или несколько мест могут ссылаться на объект `WarningRouter`. Существуют процедурные способы, ограничивающие дополнительную сложность, связанную с выполнением таких проверок на `null`. Однако при каждом обращении к методам объекта `WarningRouter` кто-то может забыть задать в исходном коде проверку на `null`, и тогда на стадии выполнения будет сгенерировано исключение `NullPointerException`.

Альтернативой использованию `null` для указания отсутствия действий служит создание класса, реализующего `WarningRouter` и ничего не делающего с предупреждающим сообщением. Этот класс показан на рис. 8.24.

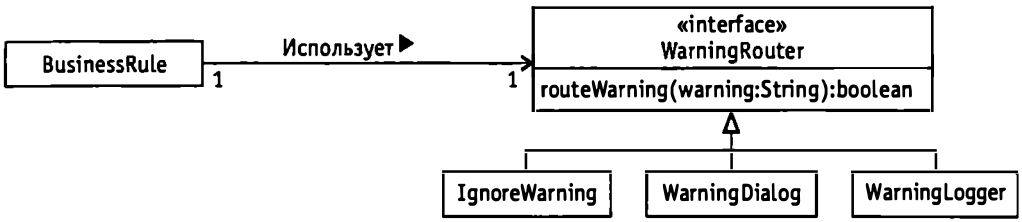


Рис. 8.24. Класс, игнорирующий предупреждение

Преимущество существования класса `IgnoreWarning` состоит в том, что его можно использовать точно так же, как и любые другие классы, реализующие интерфейс `WarningRouter`. Он не нуждается в проверке на `null` или в любой другой специальной логике.

МОТИВЫ

☺ Класс делегирует операцию другому классу. Делегирующий класс обычно не заботится о том, каким образом другой класс реализует эту операцию. Однако иногда он требует, чтобы операция была реализована как бездействующая.

☺ Нужно, чтобы класс, делегирующий операцию, делегировал ее в любом случае, включая случай отсутствия действий. Нежелательно, чтобы ситуация отсутствия действий нуждалась в каком-либо специальном коде.

РЕШЕНИЕ

На рис. 8.25 представлена диаграмма классов, демонстрирующая структуру шаблона Null Object.

Опишем роли, исполняемые классами и интерфейсом, участвующим в этом шаблоне.

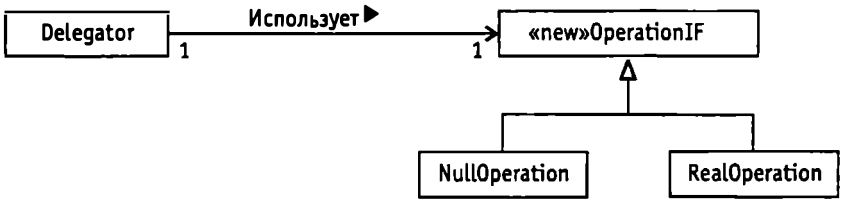


Рис. 8.25. Шаблон Null Object

Delegator. Класс в этой роли участвует в шаблоне Null Object, делегируя операцию абстрактному классу или интерфейсу. Он осуществляет такое делегирование, не принимая на себя ответственность за правильность поведения операции. Он просто полагает, что объект, которому он делегирует операцию, инкапсулирует правильное поведение даже в случае отсутствия действий.

В частности, объект в роли `Delegator` не должен выполнять проверку на `null` перед обращением к методам объекта, которому он делегирует операцию.

OperationIF. Класс в роли `Delegator` делегирует операцию интерфейсу, выступающему в этой роли. Эту роль может выполнять также абстрактный класс.

RealOperation. Классы в этой роли реализуют операцию, которую класс `Delegator` делегирует интерфейсу `OperationIF`.

NullOperation. Классы в этой роли обеспечивают бездействующую реализацию операции, которую класс `Delegator` делегирует интерфейсу `OperationIF`.

РЕАЛИЗАЦИЯ

Довольно часто экземпляры классов `NullOperation` не содержат информацию, зависящую от экземпляров. В таком случае можно сэкономить время и память, реализуя класс `NullOperation` как класс-одиночку.

СЛЕДСТВИЯ

☺ Шаблон Null Object освобождает класс, делегирующий операцию другому классу, от ответственности за реализацию бездействующей версии этой операции. Он упрощает код, который теперь не должен производить проверку на `null` перед вызовом метода, реализующего делегированную операцию. Шаблон Null Object повышает надежность кода, так как он устраняет потенциальную возможность ошибок, которые могут быть связаны с пропуском проверки на `null` в исходном тексте.

☺ Бездействующее поведение, которое инкапсулировано в классе, выступающем в роли `NullOperation`, является многократно используемым, если оно согласуется с бездействующим поведением, работающим для всех классов `Delegator`.

☹ Использование шаблона Null Object приводит к увеличению количества классов в программе. Если не сушествует интерфейса, исполняющего роль `OperationIF`, то шаблон Null Object, вводя дополнительные классы, значительно усложняет программу, несмотря на некоторое упрощение кода.

ПРИМЕР КОДА

Приведем код, реализующий классы, представленные в разделе «Контекст». Сначала — интерфейс `WarningRouter`, реализуемый классами, обеспечивающими соответствующую среде обработку предупреждающих сообщений.

```
public interface WarningRouter {
   /**
    * Этот метод передает предупреждающее сообщение
    * по любому адресу, который он считает подходящим.
    * @return Возвращает true, если тот, кто его вызвал, должен
    *     продолжить свою текущую операцию.
    */
   public boolean routeWarning(String msg) ;
} // interface WarningRouter
```

Далее приведем фрагмент кода класса `BusinessRules`, который делегирует обработку предупреждающих сообщений объектам, реализующим интерфейс `WarningRouter`.

```
class BusinessRule {
   private WarningRouter warning;
   private Date expirationDate = new Date(Long.MAX_VALUE);
   ...
   BusinessRule() {
      ...
      if (new Date().after(expirationDate)) {
         String msg = getClass().getName()+" has expired.";
         warning.routeWarning(msg);
      } // if
      ...
   } // constructor()
} // class BusinessRule
```

Теперь напишем класс, который реализует интерфейс `WarningRouter`, показывая диалоговое окно с предупреждающим сообщением.

```
class WarningDialog implements WarningRouter {
   public boolean routeWarning(String warning) {
      int r;
```

```
            r = JOptionPane.showConfirmDialog(null,
                    warning,
                    "Warning",
                JOptionPane.OK_CANCEL_OPTION,
                JOptionPane.WARNING_MESSAGE);
            return r == 0;
        } // routeWarning(String)
    } // class WarningDialog
```

Метод `routeWarning` класса `WarningDialog` возвращает true, если пользователь щелкает на кнопку OK диалогового окна, или false, если пользователь щелкает на кнопку Cancel. Следующий листинг — это класс `IgnoreWarning`. Он инкапсулирует бездействующее поведение, поэтому его метод `routeWarning` всегда возвращает true.

```
class IgnoreWarning implements WarningRouter {
    public boolean routeWarning(String warning) {
        return true;
    } // routeWarning(String)
} // class IgnoreWarning
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ NULL OBJECT

Strategy. Шаблон Null Object часто используется вместе с шаблоном Strategy.

Singleton. Если экземпляры класса `NullOperation` не содержат информации, зависящей от экземпляров, то можно сэкономить время и память, реализуя такой класс `NullOperation` как класс-одиночку.

Strategy (Стратегия)

Этот шаблон был описан в работе [GoF95].

СИНОПСИС

Связанные алгоритмы инкапсулируются в классах, реализующих общий интерфейс. Это позволяет выбирать алгоритм путем определения соответствующего класса. Кроме того, этот шаблон со временем позволяет изменять выбор алгоритма.

КОНТЕКСТ

Предположим, нужно написать программу, которая показывает календарные даты. Одно из требований, предъявляемых к программе, состоит в том, что она должна отображать праздники, отмечаемые различными нациями и религиозными группами. Пользователь должен иметь возможность выбирать те наборы праздников, которые будут отображаться на дисплее.

Это требование можно выполнить, помещая логику для каждого набора праздников в отдельный класс. В результате получится набор небольших классов, который легко можно будет дополнить новыми классами. Кроме того, желательно, чтобы классы, использующие эти классы праздников, ничего не знали ни об одном из конкретных наборов праздников. Такой проект показан на рис. 8.26.

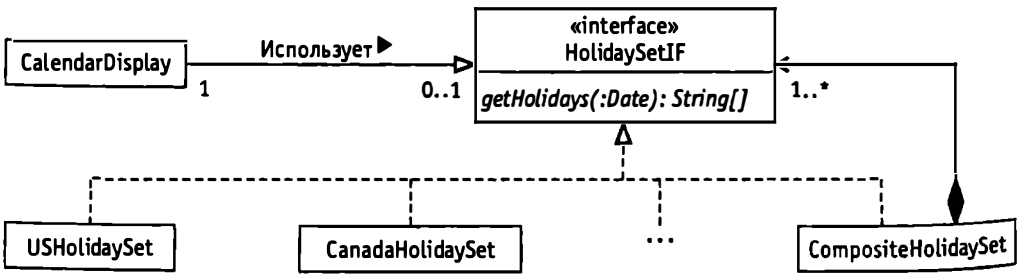


Рис. 8.26. Классы Holiday

Рассмотрим, как показанные классы работают друг с другом. Если объект `CalendarDisplay` использует в своей работе объект `HolidaySetIF`, он опрашивает этот объект, является ли отображаемый день праздничным. Такие объекты являются либо экземпляром класса типа `USHoliday`, идентифицирующего единственный набор праздников, либо экземпляром класса `CompositeHoliday`.

Класс `CompositeHoliday` работает в том случае, когда пользователю нужно отображение нескольких праздничных дат. Инстанциирование производится посредством передачи массива объектов `Holiday` его конструктору.

Такая организация позволяет объекту `CalendarDisplay` выяснять, какие праздники попадают на определенные даты, простым вызовом метода `getHolidays` объекта `HolidaySetIF`.

МОТИВЫ

☺ Программа должна обеспечивать различные варианты алгоритма или поведения.

☺ Нужно изменять поведение каждого экземпляра класса.

☺ Необходимо изменять поведение объектов на стадии выполнения.

☺ Введение интерфейса позволяет классам-клиентам ничего не знать о классах, реализующих этот интерфейс и инкапсулирующих в себе конкретные алгоритмы.

☹ Если поведение экземпляров класса не изменяется при переходе от одного экземпляра к другому или с течением времени, то самый простой выход для класса состоит в том, чтобы непосредственно реализовать эту схему своего поведения или непосредственно содержать статическую ссылку на реализующий этот алгоритм класс.

РЕШЕНИЕ

На рис. 8.27 представлена диаграмма классов, демонстрирующая классы в шаблоне Strategy.

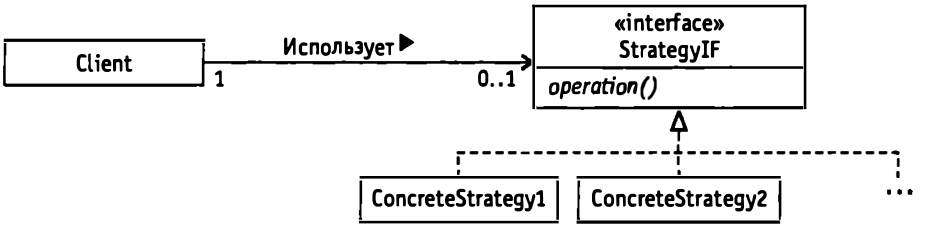


Рис. 8.27. Шаблон Strategy

Client. Класс в роли `Client` делегирует операцию интерфейсу. При этом он ничего не знает о реальном классе объекта, которому он делегирует операцию, или о том, каким образом этот класс реализует операцию.

StrategyIF. Интерфейс в этой роли обеспечивает общий способ доступа к операциям, инкапсулированным в его подклассах.

ConcreteStrategy1, ConcreteStrategy2 и т.д. Классы в этой роли предоставляют альтернативные варианты реализации операции, которую делегирует класс `Client`.

Шаблон Strategy всегда используется вместе с механизмом для определения реального объекта `ConcreteStrategy`, который будет использоваться объектом `Client`. Выбор стратегии часто обусловлен информацией о конфигурации или событиями. Реальный механизм может меняться весьма значительно, поэтому ни один конкретный механизм выбора стратегии в шаблон не включен.

РЕАЛИЗАЦИЯ

Обычно классы `ConcreteStrategy` имеют некоторые общие операции. Нужно определить такие операции в общем суперклассе.

Возможны ситуации, когда ни один из вариантов поведения, инкапсулированных в классах `ConcreteStrategy`, не является подходящим. Как правило, обработка ситуаций такого рода заключается в том, что объект `Client` должен получить `null` вместо ссылки на объект `Strategy`. Это означает необходимость проверки на `null` перед обращением к методу объекта `Strategy`. Если структура объекта `Client` не позволяет это сделать, можно использовать шаблон Null Object.

СЛЕДСТВИЯ

☺ Шаблон Strategy позволяет динамически определять поведение каждого из объектов `Client`.

☺ Шаблон Strategy упрощает классы `Client`, освобождая их от какой-либо ответственности за выбор поведения или за реализацию альтернативных вариантов поведения. Он упрощает код для объектов `Client`, устраняя операторы `if` и `switch`. В некоторых случаях он также увеличивает производительность объектов `Client`, поскольку им не нужно затрачивать какое-то время на выбор поведения.

ПРИМЕНЕНИЕ В JAVA API

Пакет `java.util.zip` содержит некоторые классы, которые используют шаблон Strategy. Оба класса — `CheckedInputStream` и `CheckedOutputStream` — используют шаблон Strategy для вычисления контрольных сумм для байтовых потоков. Эти два класса исполняют роль классов `Client`. Конструкторы обоих классов принимают в качестве аргумента интерфейс `Checksum`, выступающий в роли `AbstractStrategy`. Интерфейс `Checksum` реализуется с помощью двух классов: `Adler32` и `CRC32`, они исполняют роль `ConcreteStrategy`. На рис. 8.28 представлена диаграмма, показывающая отношения между всеми этими классами.

ПРИМЕР КОДА

Приведем код, реализующий проект, представленный в разделе «Контекст». Первый листинг содержит интерфейс `HolidaySetIF`, определяющий метод, который возвращает массив с названиями праздников, выпадающих на заданную дату. В шаблоне Strategy он выполняет роль `StrategyIF`.

```
public interface HolidaySetIF {
   public String[] NO_HOLIDAY = new String[0];
   /**
    * Возвращает массив строк, описывающих праздники, которые
    * выпадают на определенную дату.
    * Если на эту дату не выпадает никаких праздников,
    * возвращает NO_HOLIDAY.
    */
   public String[] getHolidays(Date dt) ;
} // interface HolidaySetIF
```

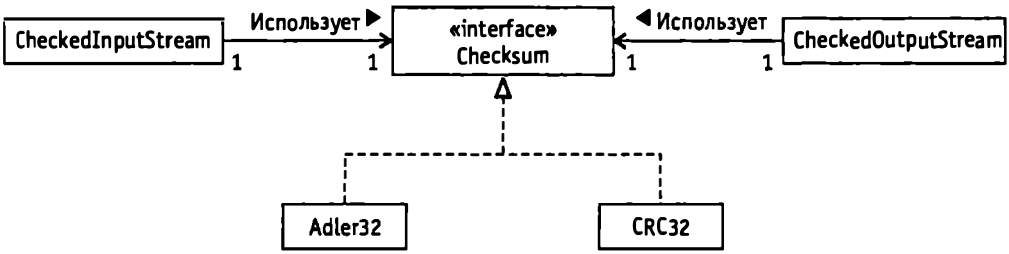


Рис. 8.28. Классы, связанные с Checksum

В интерфейсе `HolidaySetIF` объявлен массив `NO_HOLIDAY` нулевой длины, который может использоваться реализациями метода `getHolidays` как индикатор того, что нет праздников, выпадающих на некоторую дату. Это позволяет избежать создания аналогичных массивов в конкретных реализациях интерфейса `HolidaySetIF` для дней, не являющихся праздничными.

Далее следует фрагмент листинга для класса `CalendarDisplay`, который участвует в шаблоне Strategy в качестве класса `Client`.

```
class CalendarDisplay {
   private HolidaySetIF holiday;
   ...
   /**
    * Экземпляры этого закрытого класса используются
    * для кэширования информации,
```

```
     * связанной с датами, которые отображаются.
     */
    private class DateCache {
      private Date date;
      private String[] holidayStrings;

      DateCache(Date dt) {
        date = dt;
        ...
        if (holiday == null) {
          holidayStrings = holiday.NO_HOLIDAY;
        } else {
          holidayStrings = holiday.getHolidays(date);
        } // if
        ...
      } // constructor(Date)
    } // class DateCache
  } // class CalendarDisplay
```

Обратите внимание, что класс `CalendarDisplay` обязан обрабатывать возможный случай отсутствия какого-либо объекта `Holiday`, с которым он должен работать. Кроме того, этот класс полностью освобожден от любых деталей, связанных с определением праздников, выпадающих на некоторую дату.

В шаблоне Strategy в роли `ConcreteStrategy` выступают разные подклассы класса `Holiday`. Они не представляют особого интереса при рассмотрении шаблона Strategy и имеют следующую базовую структуру:

```
public class USHoliday implements HolidaySetIF {
  public String[] getHolidays(Date dt) {
    String[] holidays = HolidaySetIF.NO_HOLIDAY;
    ...
    return holidays;
  } // getHolidays(Date)
} // class USHoliday
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ STRATEGY

Adapter. С точки зрения структуры шаблон Adapter аналогичен шаблону Strategy. Отличие заключается в его предназначении. Шаблон Adapter позволяет объекту `Client` выполнять свою изначально предопределенную функцию, вызывая метод объектов, реализующих определенный интерфейс. Шаблон Strategy предос-

тавляет объекты, реализующие определенный интерфейс, с целью изменения или определения поведения объекта `Client`.

Flyweight. При наличии множества клиентских объектов лучше всего реализовывать объекты в роли `ConcreteStrategy` как объекты `Flyweight`.

Null Object. Шаблон Strategy часто используется вместе с шаблоном Null Object.

Template Method. Шаблон Template Method управляет альтернативными вариантами поведения не при помощи делегирования, а посредством создания подклассов.

Template Method (Метод шаблона)

Этот шаблон был описан в работе [GoF95].

СИНОПСИС

Создается абстрактный класс, содержащий только часть логики, необходимой для выполнения задачи. Класс организуется таким образом, что его конкретные методы вызывают абстрактный метод, в котором должна находиться пропущенная логика. Недостающая логика содержится в методах подклассов, которые замещают абстрактные методы.

КОНТЕКСТ

Предположим, нужно написать многократно используемый класс для регистрации входа пользователей в приложение или апплет. Помимо возможности многократного использования и удобства применения, к классу предъявляются следующие требования:

- возможность для пользователя ввести свои идентификатор и пароль;
- аутентифицировать ID пользователя и пароль, результатом чего должен быть объект. Если при аутентификации создается некоторая информация, которая в дальнейшем понадобится как подтверждение аутентификации, то получаемый в ходе аутентификации объект должен инкапсулировать эту информацию;
- в процессе выполнения аутентификации пользователь должен видеть изменяющееся (возможно анимационное) изображение, которое информирует пользователя о выполнении аутентификации и о том, что все идет нормально;
- уведомить остальную часть приложения или апплета о том, что вход в систему завершен, и сделать объект, создаваемый в ходе операции аутентификации, доступным для остальной части приложения.

Возможность ввода информации для пользователя и уведомление пользователя о выполняемой аутентификации не зависят от приложения. Хотя видимые пользователем строки и изображения могут быть различными для разных приложений, основаны они всегда на одной и той же логике.

Аутентификация пользователя и оповещение остальной части приложения зависят от приложения. При решении этих задач каждое приложение или апплет будет предоставлять свою собственную логику.

Способ организации класса `Logon` в значительной степени зависит от того, насколько удобно будет его использовать разработчикам. Очень гибкий механизм

представляет собой делегирование. Можно организовать класс `Logon` таким образом, что он будет просто делегировать аутентификацию пользователя и оповещение остальной части приложения. Хотя такой подход предоставляет программисту большую свободу, он не способствует принятию им правильного решения[1].

Программмисты, скорее всего, нечасто будут использовать ваш класс `Logon`. Это значит, что при его использовании они, вероятно, будут знать об этом классе немного. Подобно тому, насколько проще заполнять готовые печатные бланки, чем писать документ с нуля, задание для вашего класса `Logon` структуры помогает программистам правильно его использовать. С этой целью класс `Logon` может быть объявлен как абстрактный класс, определяющий абстрактные методы, которые соответствуют заданиям, зависящим от приложения, и для которых программист должен реализовать код. Чтобы использовать класс `Logon`, программист должен определить подклассы класса `Logon`. Поскольку методы, соответствующие заданиям, которые должны кодироваться программистом, являются абстрактными, компилятор Java будет выдавать предупреждения, если программист не будет «заполнять бланки», замещая абстрактные методы.

На рис. 8.29 представлена диаграмма, демонстрирующая такую организацию класса `Logon` и его подклассов.

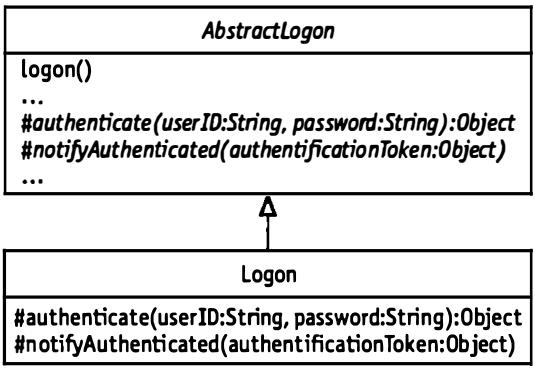


Рис. 8.29. Класс Logon и его подклассы

Класс `AbstractLogon` имеет метод под названием `logon`, содержащий общую логику для решения общих задач по регистрации входа пользователя в программу. Он вызывает абстрактные методы `authenticate` и `notifyAuthentication` для выполнения зависящих от программы заданий по аутентификации пользователя и оповещения остальной части программы о выполненной аутентификации.

---

[1] В действительности сам по себе подход не определяет правильность или неправильность решения. (*Примеч. ред.*)

МОТИВЫ

☺ Нужно спроектировать класс, который будет многократно использоваться во множестве программ.

☺ Общая структура поведения класса всегда будет одной и той же для всех приложений. Но некоторые детали поведения будут различными в разных программах, использующих этот класс.

☺ Можно упростить этот класс для тех программистов, которые не знакомы с используемым классом, и спроектировать его таким образом, чтобы программисты получали напоминание о том, что им нужно реализовать логику, зависящую от конкретной программы.

☺ Если неабстрактный класс в языке Java наследует абстрактный метод, но не замещает его, то компиляторы Java будут выдавать предупреждения.

☹ Проектирование класса таким образом, чтобы программисты реализовали логику поведения, которая зависит от конкретной программы, требует дополнительных усилий. Если класс не является часто используемым, то дополнительные усилия потрачены впустую.

РЕШЕНИЕ

Опишем общую схему поведения класса в абстрактном классе с помощью абстрактных методов. Чтобы использовать абстрактный класс, программист должен создать подкласс, который будет замещать абстрактные методы и реализовывать в них логику, зависящую от конкретного приложения.

На рис. 8.30 представлена диаграмма классов, которая демонстрирует организацию шаблона Template Method.

Рассмотрим роли, исполняемые классами в этом шаблоне.

AbstractTemplate. Класс, выступающий в этой роли, имеет конкретный метод — `templateMethod`, который содержит общую логику данного класса. Он

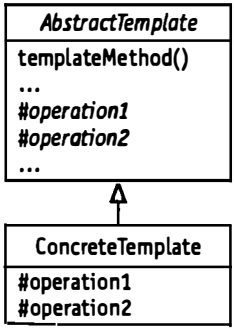


Рис. 8.30. Шаблон Template Method

вызывает другие методы, объявленные в классе AbstractTemplate как абстрактные, для активизации логики, которая различна для каждого подкласса класса AbstractTemplate.

ConcreteTemplate. Класс в этой роли представляет собой конкретный подкласс класса AbstractTemplate. В нем замещаются абстрактные методы, объявленные его суперклассом, с целью предоставления логики, необходимой для завершения логики метода templateMethod.

РЕАЛИЗАЦИЯ

Класс AbstractTemplate обеспечивает руководство, согласно которому программист обязан заместить абстрактные методы с целью предоставления логики для заполнения пробелов логики Template Method. Можно дополнить структуру, объявляя в подклассах дополнительные методы, предназначенные для замещения и обеспечивающие дополнительную или необязательную логику. В качестве примера рассмотрим класс под названием Paragraph, представляющий абзацы в документе текстового процессора.

Одна из обязанностей класса Paragraph заключается в том, что он должен определять, как заполнять строки содержащимися в них словами, и при этом оставаться в пределах заданных полей. Класс Paragraph имеет метод, который отвечает за заполнение словами абзаца. Некоторые текстовые процессоры допускают обтекание рисунков словами, поэтому класс Paragraph определяет абстрактный метод, который вызывается логикой переноса слов класса с целью определения полей для строки текста. Конкретные подклассы класса абзаца вынуждены обеспечивать реализацию этого метода с целью определения полей для каждой строки.

Некоторые текстовые процессоры включают механизм переноса слов, который автоматически определяет то место, где слова должны быть разделены для переноса. Эта характеристика позволяет переносить более длинные слова с одной строки на другую с тем, чтобы строки абзаца имели примерно одинаковую длину. Поскольку не каждый текстовый процессор будет требовать от класса Paragraph поддержку переноса слов, для этого класса нет смысла определять абстрактный метод переноса слов и заставлять подклассы замещать его. Однако было бы неплохо, если бы в классе Paragraph был бы объявлен пустой метод для переноса слов, вызываемый логикой переноса слов. Такой метод нужно объявлять для того, чтобы подклассы класса Paragraph замещали этот метод в тех случаях, когда нужна поддержка переноса слов.

Подобные методы, которые могут быть произвольно замещены с целью предоставления дополнительной или специальной функциональности, называются *hook-методами*. Чтобы программист мог знать о hook-методах, предоставляемых классом, можно применить к hook-методам соглашение об именах. Два самых распространенных соглашения, касающиеся имен для hook-методов, заключаются в том, что они либо начинаются с приставки do-, либо заканчива-

ются суффиксом -Hook. Например, придерживаясь таких соглашений об именах, метод переноса слов класса `Paragraph` может носить имя `doHyphenation` или `hyphenationHook`.

СЛЕДСТВИЕ

Программист, который пишет подкласс класса `AbstractTemplate`, вынужден замещать те методы, которые должны быть замещены с целью завершения логики суперкласса. Оптимально спроектированный класс `TemplateMethod` обладает такой структурой, которая предоставляет программисту руководство по созданию основной структуры подклассов класса `AbstractTemplate`.

ПРИМЕР КОДА

Приведем код, который реализует проект, описанный в разделе «Контекст». Сначала — код для класса `AbstractLogon`. В шаблоне Template Method он играет роль `AbstractTemplate`. Его метод шаблона называется `logon`. Метод `logon` активизирует диалоговое окно, которое предлагает пользователю ввести его идентификатор и пароль. После того как пользователь ввел ID и пароль, метод `logon` отображает окно, которое сообщает пользователю, что идет аутентификация. Это окно остается во время обращения метода `logon` к абстрактному методу `authenticate` для аутентификации ID пользователя и пароля. Если аутентификация прошла успешно, то диалоговые окна пропадают и вызывается абстрактный метод `notifyAuthentication`, чтобы известить остальные части программы о завершении аутентификации пользователя.

```
public abstract class AbstractLogon {
   /**
    * Этот метод аутентифицирует пользователя.
    * @param frame
    *     Родительское окно диалогов, которые выводит на экран
    *     этот метод.
    * @param programName Имя программы.
    */
public void logon(Frame frame, String programName) {
     Object authenticationToken;
     LogonDialog logonDialog;
     logonDialog = new LogonDialog(frame,
             "Log on to "+programName);
     JDialog waitDialog = createWaitDialog(frame);
```

Класс `LogonDialog` реализует диалог, напоминающий пользователю о вводе информации для входа. Переменная `waitDialog` ссылается на окно, содержащее сообщение для пользователя, аутентификация которого выполняется.

```
        while(true) {
            waitDialog.setVisible(false);
            logonDialog.setVisible(true);
            waitDialog.setVisible(true);
            try {
                String userID = logonDialog.getUserID();
                String password = logonDialog.getPassword();
                authenticationToken = authenticate(userID,
                    password);
                break;
            } catch (Exception e) {
                // Сообщает пользователю, что аутентификация
                // закончилась неудачей.
                JOptionPane.showMessageDialog(frame,
                    e.getMessage(),
                    "Authentication Failure",
                    JOptionPane.ERROR_MESSAGE) ;
            } // try
        }
        // Аутентификация прошла успешно.
        waitDialog.setVisible(false);
        logonDialog. setVisible(false);
        notifyAuthentication(authenticationToken);
    } // logon()
...
```

Остальная часть этого листинга просто демонстрирует абстрактные методы, определяемые в классе `AbstractLogon` и вызываемые методом `logon`.

```
    /**
     * Аутентифицирует пользователя, исходя из предоставленных
     * ID пользователя и пароля.
     * @param userID Переданное имя пользователя.
     * @param password Переданный пароль.
     * @return Возвращает объект, инкапсулирующий доказательство
     * аутентификации.
     */
    abstract protected Object authenticate(String userID,
                String password)
            throws Exception;

    /**
     * Извещает остальную часть программы о том, что пользователь
     * аутентифицирован.
```

ются суффиксом -Hook. Например, придерживаясь таких соглашений об именах, метод переноса слов класса `Paragraph` может носить имя `doHyphenation` или `hyphenationHook`.

СЛЕДСТВИЕ

Программист, который пишет подкласс класса `AbstractTemplate`, вынужден замещать те методы, которые должны быть замещены с целью завершения логики суперкласса. Оптимально спроектированный класс `TemplateMethod` обладает такой структурой, которая предоставляет программисту руководство по созданию основной структуры подклассов класса `AbstractTemplate`.

ПРИМЕР КОДА

Приведем код, который реализует проект, описанный в разделе «Контекст». Сначала — код для класса `AbstractLogon`. В шаблоне Template Method он играет роль `AbstractTemplate`. Его метод шаблона называется `logon`. Метод `logon` активизирует диалоговое окно, которое предлагает пользователю ввести его идентификатор и пароль. После того как пользователь ввел ID и пароль, метод `logon` отображает окно, которое сообщает пользователю, что идет аутентификация. Это окно остается во время обращения метода `logon` к абстрактному методу `authenticate` для аутентификации ID пользователя и пароля. Если аутентификация прошла успешно, то диалоговые окна пропадают и вызывается абстрактный метод `notifyAuthentication`, чтобы известить остальные части программы о завершении аутентификации пользователя.

```
public abstract class AbstractLogon {
   /**
    * Этот метод аутентифицирует пользователя.
    * @param frame
    *     Родительское окно диалогов, которые выводит на экран
    *     этот метод.
    * @param programName Имя программы.
    */
public void logon(Frame frame, String programName) {
      Object authenticationToken;
      LogonDialog logonDialog;
      logonDialog = new LogonDialog(frame,
              "Log on to "+programName);
      JDialog waitDialog = createWaitDialog(frame);
```

Класс `LogonDialog` реализует диалог, напоминающий пользователю о вводе информации для входа. Переменная `waitDialog` ссылается на окно, содержащее сообщение для пользователя, аутентификация которого выполняется.

```
        while(true) {
            waitDialog.setVisible(false);
            logonDialog.setVisible(true);
            waitDialog.setVisible(true);
            try {
                String userID = logonDialog.getUserID();
                String password = logonDialog.getPassword();
                authenticationToken = authenticate(userID,
                    password);
                break;
            } catch (Exception e) {
                // Сообщает пользователю, что аутентификация
                // закончилась неудачей.
                JOptionPane.showMessageDialog(frame,
                    e.getMessage(),
                    "Authentication Failure",
                    JOptionPane.ERROR_MESSAGE) ;
            } // try
        }
        // Аутентификация прошла успешно.
        waitDialog.setVisible(false);
        logonDialog. setVisible(false);
        notifyAuthentication(authenticationToken);
    } // logon()
...
```

Остальная часть этого листинга просто демонстрирует абстрактные методы, определяемые в классе `AbstractLogon` и вызываемые методом `logon`.

```
/**
 * Аутентифицирует пользователя, исходя из предоставленных
 * ID пользователя и пароля.
 * @param userID Переданное имя пользователя.
 * @param password Переданный пароль.
 * @return Возвращает объект, инкапсулирующий доказательство
 * аутентификации.
 */
abstract protected Object authenticate(String userID,
        String password)
    throws Exception;

/**
 * Извещает остальную часть программы о том, что пользователь
 * аутентифицирован.
```

```
     * @param authToken
     * Этот объект возвращает метод аутентификации.
     */
    abstract
    protected void notifyAuthentication(Object authToken) ;
} // class AbstractLogon
```

Подклассы класса `AbstractLogon` должны замещать абстрактные методы этого класса примерно таким образом:

```
public class Logon extends AbstractLogon {
    ...
    protected Object authenticate(String userID,
              String password)
              throws Exception {
       if (userID.equals("abc") && password.equals("123"))
          return userID;
       throw new Exception("bad userID");
    } // authenticate

    protected void notifyAuthentication(Object authToken) {
    } // notify(Object)
} // class Logon
```

ШАБЛОН ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЙ С ШАБЛОНОМ TEMPLATE METHOD

Strategy. Шаблон Strategy изменяет логику отдельных объектов на стадии выполнения. Шаблон Template Method изменяет логику всего класса на стадии компиляции.

Visitor (Посетитель)

Этот шаблон ранее был описан в работе [GoF95].

СИНОПСИС

Один из способов реализации операции, в которой участвуют объекты, образующие сложную структуру, заключается в том, что логика, поддерживающая эту операцию, располагается в каждом из классов таких объектов. Шаблон Visitor представляет собой альтернативный способ реализации подобных операций и позволяет избежать усложнения классов объектов структуры, помещая всю необходимую логику в отдельный класс. Этот шаблон допускает изменение логики путем использования различных классов.

КОНТЕКСТ

Предположим, в текстовый процессор нужно добавить новые характеристики, которые связаны с его способностью создавать оглавления. Это должно быть диалоговое окно, позволяющее пользователю указывать информацию, которая будет использоваться при создании оглавления. Текстовый процессор разрешает оформлять каждый абзац в соответствии с определенным стилем. С помощью диалогового окна пользователь может определить стили заголовков, которые будут вынесены в оглавление.

Текстовый процессор использует заданную в диалоговом окне информацию для построения внутренней таблицы, содержащей всю информацию, необходимую для создания многоуровневого оглавления. Далее в этом примере такая таблица будет называться внутренней таблицей оглавления (internal ToC table). Каждая строка этой таблицы содержит номер уровня, соответствующего главе, разделу, подразделу или любой другой иерархической структуре, которая задается пользователем. Кроме того, строки таблицы будут включать стиль абзаца и другие данные, необходимые для форматирования оглавления. Если в таблице указан стиль абзаца, это значит, что имеющие эти стили абзацы являются заголовками, первая строка которых будет соответствовать данному уровню оглавления.

Помимо добавления в текстовый процессор диалогового окна и внутренней таблицы оглавления, необходимо обеспечить следующие, связанные с оглавлением, характеристики:

- возможность создавать и вставлять таблицу оглавления в документ, который представлен одним файлом документа;
- возможность реорганизовать единственный файл документа и представить его в виде документа, занимающего несколько файлов, на основе уровней заголовков внутренней таблицы оглавления.

Поскольку такие операции предполагают манипулирование документом текстового процессора, любой проект, реализующий характеристики оглавления, должен будет содержать классы, используемые текстовым процессором для представления документов (рис. 8.31).

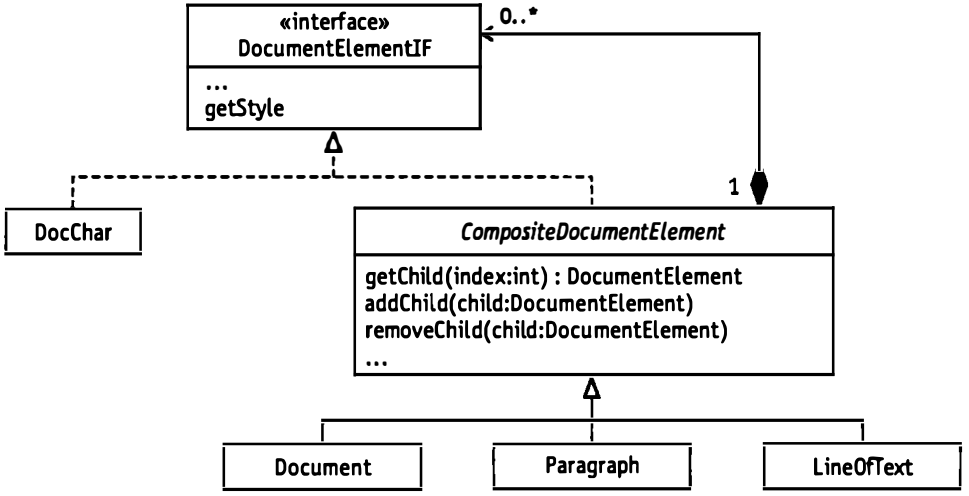


Рис. 8.31. Классы документа

С точки зрения механизма оглавления интерес представляют следующие классы: `Document`, `Paragraph` и `LineOfText`. Документ содержит абзацы, которые включают строки текста. Любой проект, предполагающий создание оглавления, должен учитывать то, что объекты `Document` могут содержать объекты, не являющиеся объектами `Paragraph`. Кроме того, необходимо принимать во внимание, что содержать объекты `Paragraph` могут не только объекты `Document`, но и другие виды объектов. И наконец, проект не должен усложнять классы, представляющие документы.

При реализации характеристик оглавления нужно рассмотреть два основных подхода. Один предусматривает размещение необходимой логики в различных классах, представляющих документ. Исходя из соображений, рассмотренных в разделе «Синопсис», это не лучшее решение.

Другой подход предполагает размещение всей логики, связанной с какой-то операцией, в одном классе. По завершении связанной с оглавлением операции объект, отвечающий за операцию, просматривает объект `Document` и содержащиеся в нем объекты. Он ищет объекты `Paragraph`, которые содержатся непосредственно в объекте `Document`. Когда он находит объекты `Paragraph` со стилем, который хранится во внутренней таблице оглавления, управляющей связанной с оглавлением операцией, он предпринимает необходимые действия (рис. 8.32). Структура, показанная на рис. 8.32, использует также классы, изображенные на рис. 8.31.

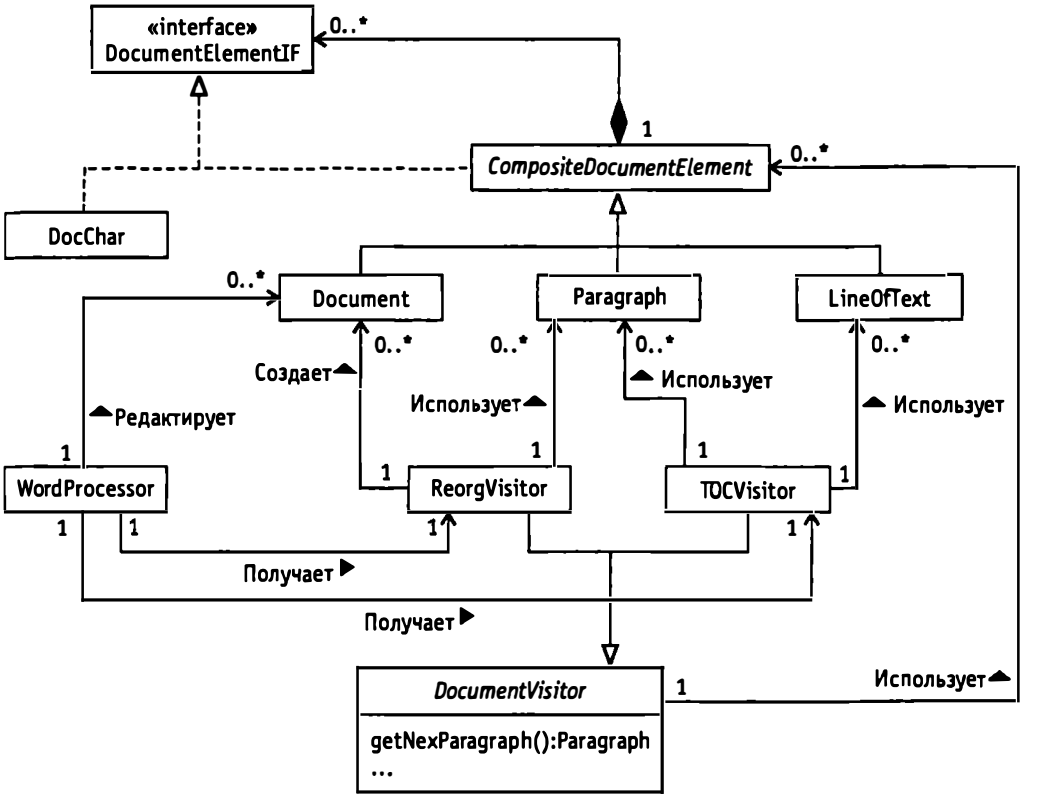


Рис. 8.32. Классы оглавления

Рассмотрим следующие классы.

WordProcessor. Класс `WordProcessor` отвечает за создание и редактирование объектов, представляющих документ. Для редактирования документа он использует другие классы, указанные на диаграмме.

DocumentVisitor. Это абстрактный класс. Его подклассы исследуют объекты, образующие документ, с целью создания оглавления или реорганизации документа для представления его в виде нескольких файлов. Класс `DocumentVisitor` обеспечивает логику, используемую его подклассами для навигации по этим объектам.

Идея такова, что экземпляры подклассов класса `DocumentVisitor` «посещают» (обращаются к) объекты, входящие в состав документа, собирают и обрабатывают информацию от каждого объекта.

TOCVisitor. Этот подкласс класса `DocumentVisitor` отвечает за создание оглавления. Он просматривает каждый объект `Paragraph`, которым непосредственно владеет объект `Document`. Когда он находит объект `Paragraph`, который имеет стиль, хранящийся во внутренней таблице оглавления, он создает соот-

ветствующий элемент оглавления. Элемент оглавления использует содержимое первого объекта `LineOfText`, принадлежащего объекту `Paragraph`.

ReorgVisitor. Этот подкласс класса `DocumentVisitor` отвечает за автоматическое разделение документа на несколько файлов. Сначала ему сообщают, чтобы он осуществил поиск абзацев, соответствующих определенному уровню структуры документа. Он находит этот уровень структуры во внутренней таблице оглавления и считывает из нее стиль, связанный с этим уровнем структуры. Затем он просматривает все объекты `Paragraph`, принадлежащие объекту `Document`. Он ищет те объекты `Paragraph`, которые имеют стиль, прочитанный им из таблицы. Когда `ReorgVisitor` находит объект `Paragraph` с нужным стилем, он создает новый объект `Document`, перемещает найденный объект `Paragraph` вместе со всеми объектами `Paragraph`, непосредственно следующими за ним и находящимися на более низком уровне структуры, во вновь созданный объект `Document`. Класс `ReorgVisitor` записывает в файл новый объект `Document` и все объекты абзацев, связанные теперь с ним. Он заменяет объекты `Paragraph`, которые он переместил из исходного объекта `Document`, новым объектом, содержащим имя файла, в котором теперь хранятся перемещенные абзацы.

МОТИВЫ

☺ Имеются различные операции, которые должны выполняться для некоторой структуры объектов.

☺ Структуру образуют объекты, принадлежащие разным классам.

☺ Типы объектов структуры меняются не часто, а их объединения являются согласованными и предсказуемыми.

РЕШЕНИЕ

В этом разделе рассматриваются две версии шаблона Visitor. Первая представляет идеальное решение, дающее идеальный результат. К сожалению, во многих ситуациях идеальное решение не будет работать или будет работать неэффективно. Вторая версия шаблона Visitor применима для множества разных ситуаций, но связана с издержками из-за ввода дополнительных зависимостей между классами.

На рис. 8.33 представлена диаграмма классов, демонстрирующая роли, исполняемые классами в идеальной версии шаблона Visitor.

Опишем роли, исполняемые такими классами.

Client. Экземпляры классов в этой роли предназначены для манипулирования всей структурой объектов и объектами, входящими в состав этой структуры. С целью выполнения вычислений для структур объектов, за которые они отвечают, они используют объекты `ConcreteVisitor`.

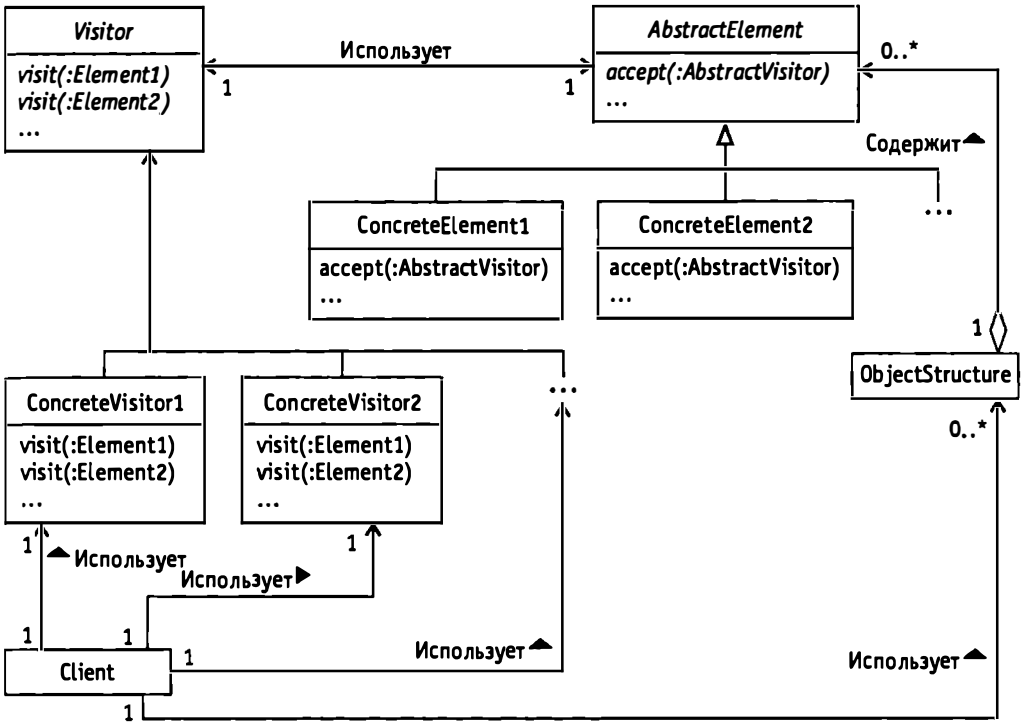


Рис. 8.33. Идеальная версия шаблона Visitor

ObjectStructure. Экземпляр класса в этой роли служит корневым объектом структуры объектов. «Посещая» объекты некоторой структуры, объект `Visitor` начинает с экземпляра класса `ObjectStructure`, а затем перемещается по другим объектам этой структуры.

Некоторый класс может участвовать в шаблоне Visitor в роли `ObjectStructure`, а также исполнять роль `ConcreteElement` (в качестве элемента структуры объектов).

AbstractElement. Класс в этой роли представляет собой абстрактный суперкласс для объектов, которые входят в состав структуры объектов. Он определяет абстрактный метод, который на рис. 8.33 указан под названием `accept`. В качестве аргумента он принимает объект `AbstractVisitor`. Подклассы класса `AbstractElement` обеспечивают реализацию метода `accept`, который вызывает метод объекта `AbstractVisitor`, а затем передает объект `AbstractVisitor` методу `accept` других объектов `AbstractElement`.

ConcreteElement1, ConcreteElement2 и т.д. Экземпляры классов в этой роли представляют собой элементы структуры объектов. При выполнении вычислений для объектов структуры объект `AbstractVisitor` передается методу `accept` объекта `ConcreteElement`. Метод `accept` передает объект `ConcreteElement` одному из методов объекта `AbstractVisitor`, поэтому он может включить

в свои вычисления объект ConcreteElement. По окончании операции объект ConcreteElement передает объект AbstractVisitor методу accept другого объекта ConcreteElement.

Visitor. Выступающий в этой роли класс представляет собой абстрактный суперкласс для классов, выполняющих вычисления над элементами структуры объектов. Он определяет метод для каждого класса, который будут «посещать» его подклассы, поэтому их экземпляры могут передавать сами себя объектам Visitor и принимать участие в вычислениях.

ConcreteVisitor1, ConcreteVisitor2 и т.д. Экземпляры классов, выступающих в этой роли, образуют структуру объектов.

На рис. 8.34 представлена диаграмма взаимодействия, более очевидно демонстрирующая то, как объекты Visitor сотрудничают с объектными структурами.

На этом рисунке показано взаимодействие между объектом Visitor и элементами структуры объектов. После того как объект Visitor представлен объекту ObjectStructure, объект ObjectStructure передает объект Visitor методу accept объекта ConcreteElement. Объект ConcreteElement передает сам себя методу visit объекта Visitor, что позволяет этому объекту включить объект Visitor в свои вычисления. Затем объект ConcreteElement передает объект Visitor другому объекту ConcreteElement, поэтому объект Visitor может «посетить» его. При передаче объекта Visitor другим объектам ConcreteElement цикл продолжается. Объект ConcreteElement может быть связан с любым количеством других объектов ConcreteElement. Он может передавать объект Visitor нескольким, всем или ни одному связанному с ним объекту ConcreteElement.

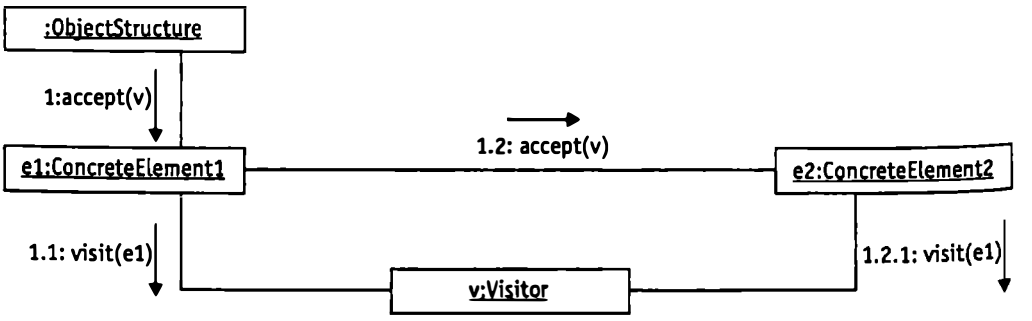


Рис. 8.34. Взаимодействие в идеальной версии шаблона Visitor

В этой версии шаблона Visitor объекты AbstractElement определяют, какие элементы структуры объектов «посещают» объект Visitor и каков порядок этих «посещений». Данная версия хорошо работает в тех случаях, когда объекты Visitor любого вида, «посещая» элементы структуры объектов, следуют по одному и тому же маршруту. Преимущество здесь состоит в том, что классы Visitor поддерживаются независимо от способа организации структуры объ-

ектов. Однако в некоторых ситуациях эта версия не работает. Это следующие ситуации.

- Появляются посетители, изменяющие структуру объектов. Описанный в разделе «Контекст» пример объекта `Visitor`, разделяющего документ на несколько файлов, — это именно такая ситуация.
- Структуры объектов слишком велики, и, чтобы «посетить» все объекты, объекту `Visitor` потребуется недопустимо много времени, тогда как он должен «посетить» только некоторые из объектов структуры.

На рис. 8.35 представлена диаграмма классов для другой версии шаблона Visitor. В этой версии шаблона Visitor классы `Visitor` отвечают за навигацию по структуре объектов в соответствии со своим собственным маршрутом. Организованные таким образом посетители способны изменять структуру объектов или выборочно перемещаться по ней. Недостатком подобной организации является то, что классы `Visitor` не могут быть многократно используемыми, поскольку для перемещения по структуре объектов они должны делать некоторые предположения, касающиеся этой структуры.

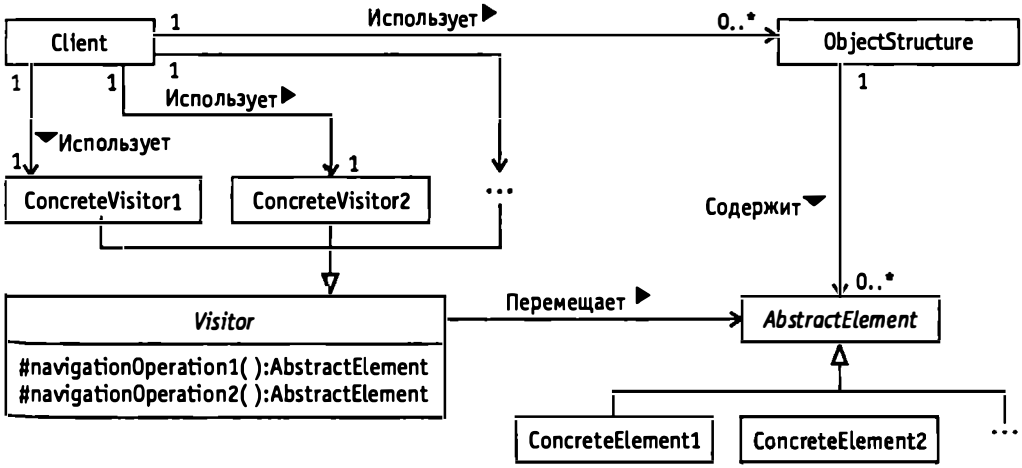


Рис. 8.35. Неидеальная версия шаблона Visitor

Класс `AbstractElement` в этой версии шаблона не содержит каких-либо методов, специально связанных с объектами `Visitor`. Вместо этого класс `Visitor` определяет методы, используемые его подклассами для навигации по структуре объектов.

РЕАЛИЗАЦИЯ

При реализации шаблона Visitor прежде всего нужно решить, можно ли использовать идеальную версию шаблона. Если да, то следует использовать именно эту версию шаблона Visitor, так как ее реализация и сопровождение требует

меньших усилий. Если использование идеальной версии шаблона Visitor невозможно, помещают в класс `Visitor` (а не в его подклассы) как можно большую часть логики, предназначенную для навигации по структуре объектов. Тогда число зависимостей таких объектов `ConcreteVisitor` от структуры объектов будет минимальным, и сопровождение станет более простым.

СЛЕДСТВИЯ

☺ Шаблон Visitor позволяет легко добавлять в структуру объектов новые операции. Классы `ConcreteElement` не зависят от классов `Visitor`, поэтому добавление нового класса `Visitor` не требует изменения класса `AbstractElement` или любого из его подклассов.

☺ Шаблон Visitor помещает логику, необходимую для выполнения некоторой операции, в один сцепленный класс — `ConcreteVisitor`. Такая схема сопровождается легче по сравнению с операциями, распределенными по многим классам `ConcreteElement`.

☺ Единственный объект `Visitor` сохраняет состояние, необходимое для выполнения операции над структурами объектов. Такая схема легче сопровождается и более эффективна по сравнению со схемой, при которой информация о состоянии должна передаваться в виде дискретных значений от одного объекта другому.

☹ Другое следствие использования шаблона Visitor состоит в том, что нужны дополнительные усилия для добавления новых классов `ConcreteElement`. Идеальная версия шаблона Visitor требует от программиста добавления нового метода `visit` в каждый класс `ConcreteVisitor` для каждого добавляемого класса `ConcreteElement`. Для другой версии шаблона Visitor может понадобиться изменение логики, используемой классами `Visitor` для навигации по структуре объектов.

☹ Прямым следствием шаблона Visitor является то, что классы `ConcreteElement` должны обеспечивать предоставление о себе столько информации, сколько должно быть достаточно для проведения вычислений объектами `Visitor`. Это может означать, что программист открывает информацию, которая в противном случае была бы скрыта инкапсуляцией класса.

ПРИМЕР КОДА

Приведем коды для некоторых классов, представленных в проекте по созданию оглавления, описанном в разделе «Контекст». Сначала код для класса `WordProcessor`, содержащего общую логику для текстового процессора. Он отвечает за инициирование операций, манипулирующих документами.

```
public class WordProcessor {
   // Редактируемый в данный момент документ.
```

```
    private Document activeDocument;
...
    /**
     * Реорганизуем документ, представляя его в виде нескольких
     * файлов.
     */
    private void reorg(int level) {
        new ReorgVisitor(activeDocument, level);
    } // reorg(int)

    /**
     * Строим оглавление.
     */
    private TOC buildTOC() {
        return new TOCVisitor(activeDocument).buildTOC();
    } // buildTOC()
} // class WordProcessor
```

Следующий листинг — класс DocumentVisitor. Класс DocumentVisitor представляет собой абстрактный суперкласс для классов, реализующих операции, в ходе которых «посещаются» многие объекты, входящие в состав документа.

```
abstract class DocumentVisitor {
    private Document document;
    private int docIndex = 0; // Индекс, используемый
                              // для навигации потомков документа.
    DocumentVisitor(Document document) {
        this.document = document;
    } // constructor(Document)

    protected Document getDocument() { return document; }

    /**
     * Возвращает следующий абзац, который является
     * непосредственной частью документа.
     */
    protected Paragraph getNextParagraph() {
        Document myDocument = document;
        while (docIndex < myDocument.getChildCount()) {
            DocumentElementIF docElement;
```

```
            docElement = myDocument.getChild(docIndex);
            docIndex += 1;
            if (docElement instanceof Paragraph)
               return (Paragraph)docElement;
         }
         return null;
      } // getNextParagraph()
...
} // class DocumentVisitor
```

Следующий листинг — это класс `ReorgVisitor`, который должен «посещать» абзацы документа и перемещать в отдельный документ те из них, которые относятся к заданному уровню структуры оглавления.

```
class ReorgVisitor extends DocumentVisitor {
   private TocLevel[] levels;

   ReorgVisitor(Document document, int level) {
      super(document);
      this.levels = document.getTocLevels();
      Paragraph p;
      while ((p = getNextParagraph()) != null) {
         ...
      } // while
   } // constructor(Document)
} // class ReorgVisitor
```

В последнем листинге описан класс `TOCVisitor`. Класс `TOCVisitor` отвечает за создание оглавления. Он более глубоко проникает в структуру объектов документа, манипулируя при этом объектами `Document`, `Paragraph` и `LineOfText`. Он использует объект `LineOfText`, потому что элемент оглавления будет содержать текст из первого объекта `LineOfText` из абзаца, которому соответствует элемент оглавления.

```
class TOCVisitor extends DocumentVisitor {
   private Hashtable tocStyles = new Hashtable();

   TOCVisitor(Document document) {
      super(document);
      TocLevel[] levels = document.getTocLevels();
      // Помещаем стили в хэш-таблицу.
      for (int i=0; i < levels.length; i++) {
         tocStyles.put(levels[i].getStyle(), levels[i]);
      } // for
   } // constructor(Document)
```

```
    TOC buildTOC() {
       TOC toc = new TOC();
       Paragraph p;
       while ((p = getNextParagraph()) != null) {
          String styleName = p.getStyle();
          if (styleName != null) {
             TocLevel level;
             level = (TocLevel)tocStyles.get(styleName);
             if (level != null) {
                LineOfText firstLine = null;
                for (int i = 0; i < p.getChildCount();i++){
                DocumentElementIF e = p.getChild(i);
                if (e instanceof LineOfText) {
                   firstLine = (LineOfText)e;
                   break;
                } // if
                ...
                } // for
             } // if
          } // if
       } // while
       return toc;
    } // buildTOC()
 } // class TOCVisitor
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ VISITOR

Iterator. Шаблон Iterator представляет собой альтернативу шаблону Visitor в том случае, когда структура объектов, по которой осуществляется навигация, является линейной.

Little Language. Шаблон Visitor можно использовать в шаблоне Little Language для реализации некоторой его части.

Composite. Шаблон Visitor часто используется вместе с такими структурами объектов, которые организованы при помощи шаблона Composite.

ЛАВА 9

Шаблоны проектирования для конкурирующих операций

редставленные в данной главе шаблоны предназначены для координирова-ия конкурирующих операций и должны решать главным образом проблемы ух разных типов.

Совместно используемые ресурсы. Если конкурирующие операции обращаются к одним и тем же данным или другому общему ресурсу, они могут конфликтовать друг с другом, если эти операции осуществляют доступ к ресурсу в один и тот же момент. Чтобы обеспечить правильное выполнение таких операций, их нужно ограничить таким образом, чтобы доступ к общему ресурсу в какой-то момент получала только одна операция. Однако чрезмерное ограничение операций может привести к их взаимной блокировке и невозможности выполнения.

Взаимная блокировка (deadlock) — это ситуация, когда одна операция, прежде чем продолжить выполнение, ожидает действие второй операции. Если каждая операция ожидает действий со стороны другой, то ожидание затягивается и ничего не происходит. Иногда взаимную блокировку называют *смертельным объятием* (deadly embrace).

2. **Последовательность операций.** Если операции вынуждены получать доступ к общему ресурсу одновременно, может возникнуть необходимость в том, чтобы они обращались к совместному ресурсу в определенном порядке. Например, объект не может быть удален из структуры данных до того, как он должен быть добавлен в эту структуру данных.

Шаблон Single Threaded Execution — это самый важный шаблон данной главы. Большая часть вопросов, относящихся к совместно используемым ресурсам, может быть решена только при помощи шаблона Single Threaded Execution, который обеспечивает доступ к ресурсу в некоторый момент только одному потоку. Ситуации, в которых имеет значение порядок выполнения операций, являются менее распространенными, и они могут обрабатываться с помощью шаблона Scheduler.

Шаблон Guarded Suspension предоставляет руководство к действию в том случае, когда поток имеет монопольный доступ к ресурсу и оказывается, что он не может завершить выполнение операции над этим ресурсом, поскольку что-то еще не готово.

Если некоторая операция нуждается в монопольном доступе к нескольким ресурсам, шаблон Lock Object позволяет упростить координацию доступа к нескольким ресурсам.

Шаблон Balking удобен в том случае, когда операция должна быть выполнена либо немедленно, либо никогда.

Шаблон Read/Write Lock обеспечивает альтернативный вариант доступа вида «один поток в какой-то момент», если одни операции могут совместно использовать один и тот же ресурс, а другие операции этого делать не могут.

Шаблон Producer-Consumer координирует объекты, создающие некоторый ресурс, и объекты, использующие этот ресурс.

Шаблон Two-Phase Termination применяется для правильного последовательного закрытия потоков.

Шаблон Double Buffering представляет собой специальную версию шаблона Producer-Consumer. Он позволяет создавать необходимые ресурсы заранее, т.е. до того как они понадобятся.

Шаблон Asynchronous Processing позволяет избежать ожидания результатов операции, если этот результат не нужен немедленно.

Шаблон Future позволяет классам, вызывающим операцию, не знать о том, синхронной или асинхронной является эта операция.

Single Threaded Execution (Однопоточное выполнение)

Шаблон Single Threaded Execution известен также под названием Critical Section.

СИНОПСИС

Обращение некоторых методов к данным или другим общим ресурсам может привести к ошибочным результатам, если имеют место конкурентные вызовы метода, желающие получить доступ к данным или другому ресурсу одновременно. Для решения этой проблемы шаблон Single Threaded Execution препятствует конкурентным вызовам метода, тем самым запрещая параллельное выполнение этого метода.

КОНТЕКСТ

Предположим, что создается ПО для системы, которая контролирует движение транспорта на главном шоссе. В стратегических точках, расположенных на шоссе, датчики фиксируют количество машин, прошедших в течение одной минуты. Датчики отправляют информацию на главный компьютер, который управляет электронными знаками, расположенными вблизи главных дорожных развязок. Эти знаки выводят сообщения для водителей, информируя их об условиях на дорогах с тем, чтобы водители могли выбрать альтернативные маршруты.

Датчики измеряют поток машин, располагаясь на каждой полосе дороги. Все датчики присоединены к контроллеру, который суммирует количество машин, прошедших данное место дороги в течение каждой минуты. Контроллер связан с передатчиком, который посылает на главный компьютер все итоговые суммы, представляющие количество машин, прошедших за одну минуту. На рис. 9.1 представлена диаграмма классов, описывающая эти отношения.

Рассмотрим эти классы.

TrafficSensor. Каждый экземпляр этого класса соответствует физическому сенсорному устройству. Каждый раз, когда машина проходит мимо физического сенсорного устройства, соответствующий экземпляр класса `TrafficSensor` вызывает метод `vehiclePassed` класса `TrafficSensorController`. Каждый объект `TrafficSensor` выполняется в своем собственном потоке, что позволяет управлять входными данными от связанного с ним сенсора асинхронно по отношению к другим сенсорам.

TrafficTransmitter. Экземпляры этого класса отвечают за передачу данных о количестве машин, прошедших через некоторое место на дороге в течение одной минуты. Объект `TrafficTransmitter` получает величину, обозначающую

количество машин, прошедших определенное место дороги, вызывая метод `getAndClearCount` соответствующего объекта `TrafficSensorController`. Метод `getAndClearCount` объекта `TrafficSensorController` возвращает число, обозначающее количество машин, прошедших мимо сенсора с момента предыдущего вызова метода `getAndClearCount`.

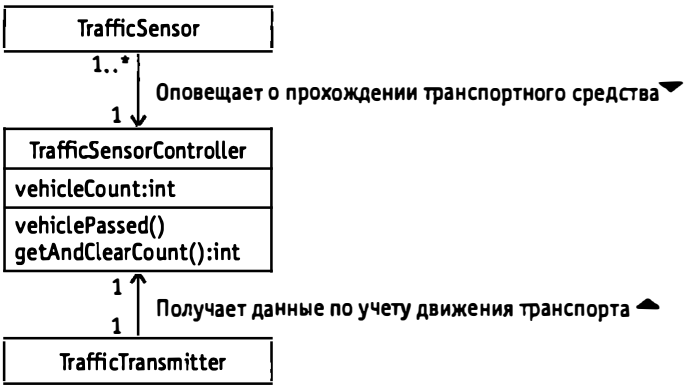


Рис. 9.1. Классы датчика движения транспорта

TrafficSensorController. Экземпляры класса `TrafficSensor` и класса `TrafficTransmitter` вызывают методы класса `TrafficSensorController` для обновления, считывания и очистки переменной, содержащей количество машин, прошедших через некоторое место дороги.

Возможна ситуация, когда два объекта `TrafficSensor` вызывают метод `vehiclePassed` объекта `TrafficSensorController` одновременно. Если оба вызова будут выполняться в одно и то же время, то это приведет к ошибке. Предполагается, что каждое обращение к методу `vehiclePassed` должно увеличивать значение счетчика машин на единицу. Однако, если в один и тот же момент времени выполняются два обращения к методу `vehiclePassed`, счетчик машин увеличивается не на два, а на единицу. Опишем последовательность событий для случая одновременного выполнения обоих обращений к методу.

1. Оба вызова одновременно считывают одно и то же значение `vehicleCount`.
2. Оба вызова добавляют единицу к одной и той же величине.
3. Оба вызова сохраняют одно и то же значение в `vehicleCount`.

Очевидно, что, допуская одновременное выполнение нескольких обращений к методу `vehiclePassed`, в результате получим число, которое меньше реального количества прошедших машин. Хотя небольшое занижение количества машин не может представлять серьезной угрозы для этого приложения, существует похожая проблема, которая является более серьезной.

Объект `TrafficTransmitter` периодически вызывает метод `getAndClearCount` объекта `TrafficSensorController`. Метод `getAndClearCount` считывает зна-

чение переменной vehicleCount объекта TrafficSensorController, а затем сбрасывает ее в ноль. Если методы vehiclePassed и getAndClearCount объекта TrafficSensorController вызываются одновременно, создается ситуация, которая называется состязание.

Состязание (race condition) — это ситуация, исход которой зависит от порядка завершения конкурирующих операций. Если последним завершается метод getAndClearCount, то он устанавливает значение переменной vehicleCount в ноль, удаляя результат вызова метода vehiclePassed. Это просто другой способ неправильного подсчета машин, когда значение счетчика меньше реального количества машин. Однако проблема становится более серьезной, если последним финиширует метод vehiclePassed.

Если последним завершается метод vehiclePassed, он заменяет нулевое значение, установленное методом getAndClearCount, своим значением, равным прочитанной им величине плюс единица. Это означает, что при следующем вызове метод getAndClearCount возвратит значение, которое содержит также машины, учтенные перед предыдущим обращением к методу getAndClearCount. Количество машин, содержащееся в счетчике, может в два, три и даже более раз превышать их реальное количество.

Большое значение счетчика может привести к тому, что центральный компьютер сделает вывод, что началась дорожная «пробка» и что он должен показать на электронных знаках сообщения, предлагающие водителям следовать по другим маршрутам. Тогда ошибка подобного рода действительно может стать причиной возникновения дорожной «пробки».

Если потребовать, чтобы в какой-то момент только один поток выполнял метод vehiclePassed или метод getAndClearCount объекта TrafficSensorController, это будет простым решением проблемы. Можно указать такое проектное решение, обозначив параллельность этих методов как охраняемую (guarded). В схемах UML обозначение охраняемой параллельности метода эквивалентно объявлению его в Java как синхронизированного (рис. 9.2).

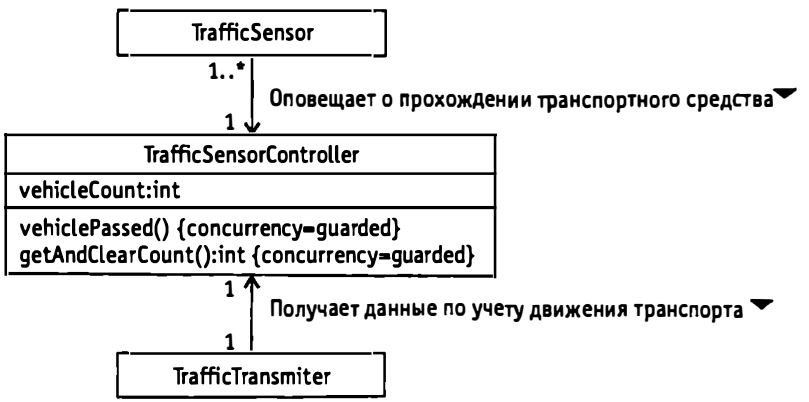


Рис. 9.2. Синхронизированные классы сенсоров движения транспорта

Сколько угодно потоков могут обращаться к охраняемым методам одного и того же объекта одновременно. Однако в какой-то момент только одному потоку разрешено выполнение охраняемых методов объекта. Пока один поток выполняет охраняемые методы объекта, другие потоки ожидают до тех пор, пока поток не завершит выполнение этих методов. Затем для выполнения следующих охраняемых методов объекта будет выбран произвольным образом другой поток из числа ожидающих. Такой механизм гарантирует однопоточное выполнение охраняемых методов объекта.

МОТИВЫ

- ☺ Класс содержит методы, которые обновляют или задают значения в переменных экземпляра или переменных класса.
- ☺ Метод манипулирует внешними ресурсами, которые поддерживают только одну операцию в какой-то момент времени.
- ☺ Методы класса могут вызываться параллельно различными потоками.
- ☺ Не существует временно́го ограничения, которое требовало бы от метода немедленного выполнения, как только его вызывают.

РЕШЕНИЕ

Гарантировать, чтобы операции, параллельное выполнение которых даст ошибочный результат, не выполнялись бы параллельно. С этой целью методы, не подлежащие параллельному выполнению, делаются охраняемыми (рис. 9.3).

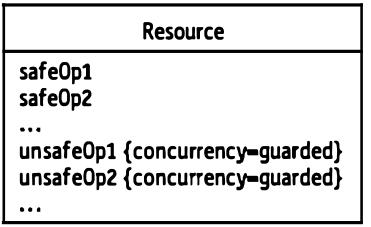


Рис. 9.3. Шаблон Single Threaded Execution

Класс, изображенный на рис. 9.3, имеет методы двух видов. Первый вид — неохраняемые методы `safeOp1`, `safeOp2` и т.д., которые могут безопасно конкурентно выполняться. Второй вид — охраняемые методы `unsafeOp1`, `unsafeOp2` и т.д., которые не могут безопасно параллельно выполняться. Если различные потоки одновременно вызывают охраняемые методы объекта `Resource`, в какой-то момент времени только одному потоку разрешено выполнять такой метод. Остальные потоки ждут, пока этот поток не закончит свою работу.

РЕАЛИЗАЦИЯ

В языке Java охраняемые методы реализуются посредством объявления их синхронизированными. Для вызова синхронизированного метода может потребоваться больше времени, чем для вызова несинхронизированного. Рассмотрим диаграмму взаимодействия (рис. 9.4).

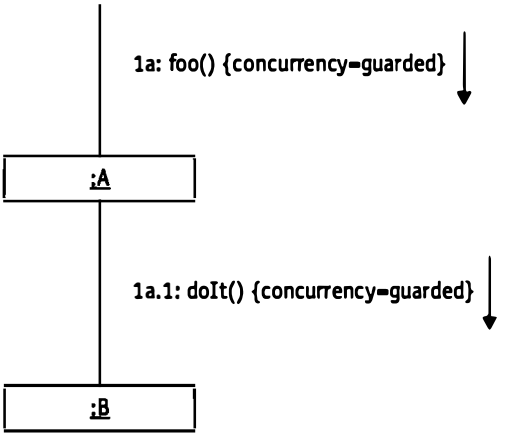


Рис. 9.4. Факторинг синхронизации

Синхронизированный метод класса `A` вызывает метод `doIt` класса `B`. Метод `doIt` является синхронизированным. Если метод `doIt` вызывается только синхронизированными методами класса `A`, то можно произвести оптимизацию и сделать метод `doIt` несинхронизированным. Он будет выполняться только одним потоком в какой-то момент времени, поскольку он вызывается только такими методами, выполнение которых предусматривает участие только одного потока в некоторый момент времени.

Такая оптимизация называется факторингом синхронизации. *Факторинг синхронизации* (synchronisation factoring) — это небезопасная оптимизация в том смысле, что, если программа будут изменена так, что к методу `doIt` будут производиться параллельные обращения, программа перестанет правильно функционировать. Поэтому, если разработчик решает, что оптимизацию стоит проводить «вручную», он должен внести комментарии в диаграммы проекта и исходный текст кода, чтобы предупредить программистов о выполненной оптимизации.

Существуют компиляторы и JVM, которые выполняют подобную оптимизацию автоматически: например, JVM, применяющие технологию HotSpot фирмы Sun. Существуют компиляторы, которые могут осуществлять факторинг синхронизации в определенных случаях. При использовании JVM или компилятора, который делает оптимизацию автоматически, лучше не делать подобную оптимизацию вручную.

СЛЕДСТВИЯ

☺ Если у класса есть методы, имеющие небезопасный доступ к переменным или другим ресурсам, то все эти методы делаются охраняемыми, обеспечивая таким образом безопасность потоков.

• Если охраняемыми делаются методы, которые в этом не нуждаются, производительность может быть снижена. Это объясняется двумя причинами. Одна причина заключается в том, что могут существовать некоторые дополнительные затраты просто на вызов охраняемого метода. Другая причина состоит в том, что если метод объявлен охраняемым, а этого можно было бы и не делать, то вызвавшая его сторона может быть блокирована (хотя этого можно было бы избежать).

☹ Если методы делаются охраняемыми, становится возможной взаимная блокировка потоков. Она возникает в том случае, когда каждый из двух потоков должен монопольно использовать ресурс и каждый из потоков, прежде чем продолжить, ожидает, пока другой не освободит свой ресурс. Каждый поток ждет ресурс, к которому другой поток уже имеет монопольный доступ, и поэтому оба потока будут ждать бесконечно, не получая доступа к ожидаемому ресурсу.

Рассмотрим пример, представленный на рис. 9.5. Поток `1a` вызывает метод `foo` объекта `x`, и одновременно поток `1b` вызывает метод `bar` объекта `y`. Затем поток `1a` вызывает метод `bar` объекта `y` и ждет, пока поток `1b` закончит свое обращение к этому методу. Поток `1b` вызывает метод `foo` объекта `x` и ждет, пока поток `1a` не завершит свое обращение к этому методу. С этого момента два потока взаимно заблокированы. Каждый ждет, пока другой завершит свой вызов.

Взаимно заблокированными могут быть не только два, а несколько потоков.

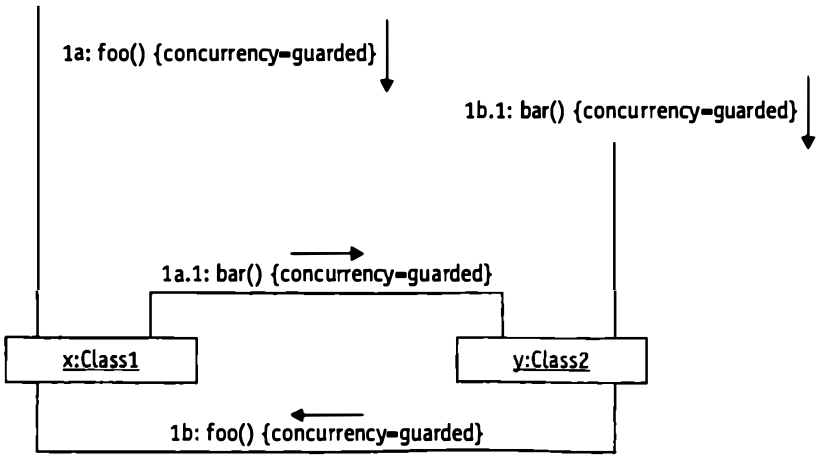


Рис. 9.5. Взаимная блокировка

ПРИМЕНЕНИЕ В JAVA API

Многие методы класса `java.util.Vector` являются синхронизированными с целью обеспечения последовательного доступа к внутренним структурам данных объектов `Vector`.

ПРИМЕР КОДА

Приведем код, реализующий проект для сенсора движения транспорта, рассмотренный в разделе «Контекст». Сначала приводится код для класса `TrafficSensor`. Экземпляры класса `TrafficSensor` связаны с сенсором движения транспорта, который фиксирует прохождение мащиной некоторого места на полосе движения. В этот момент вызывается метод `detect` экземпляра, отвечающий за оповещение всех заинтересованных объектов о прохождении транспортного средства.

```
public class TrafficSensor implements Runnable {
   private TrafficObserver observer;

   /**
    * Constructor
    * @param observer
    *     Объект, предназначенный для оповещения о том, что сенсор
    *     движения транспорта, связанный с  этим объектом,
    *     обнаруживает проходящую машину.
    */
   public TrafficSensor(TrafficObserver observer) {
      this.observer = observer;
      new Thread(this).start();
   } // constructor(TrafficObserver)

   /**
    * Общая логика для потока этого объекта.
    */
   public void run() {
      monitorSensor();
   } // run()
   // Этот метод вызывает метод detect объекта, когда
   // связанный с ним сенсор движения транспорта фиксирует
   // проходящую машину.
   private native void monitorSensor() ;

   // Этот метод вызывается методом monitorSensor,
   // чтобы сообщить о прохождении транспортного средства
   // наблюдателю этого объекта.
```

```
    private void detect() {
      observer.vehiclePassed();
    } // detect()
...
    /**
     * Классы должны реализовывать этот интерфейс,
     * чтобы объект TrafficSensor мог сообщить им о прохождении
     * машин.
     */
    public interface TrafficObserver {
    /**
      * Вызывается тогда, когда TrafficSensor фиксирует
      * проходящее транспортное средство.
      */
      public void vehiclePassed();
    } // interface TrafficObserver
  } // class TrafficSensor
```

Следующий класс — `TrafficTransmitter`. Экземпляры класса `TrafficTransmitter` отвечают за передачу значения, определяющего количество машин, прошедших через данное место дороги за одну минуту.

```
  public class TrafficTransmitter implements Runnable {
    private TrafficSensorController controller;
    private Thread myThread;

    /**
     * constructor.
     * @param controller От TrafficSensorController этот объект
     * будет получать значение счетчика количества прошедших
     * машин.
     */
    public
    TrafficTransmitter(TrafficSensorController controller) {
      this.controller = controller;
      ...
      myThread = new Thread(this);
      myThread.start();
    } // constructor(TrafficSensorController)

    /**
     * Передает значение счетчика количества машин, прошедших
     * за одну минуту.
     */
```

```
    public void run() {
      while (true) {
        try {
            myThread.sleep(60*1000);
            transmit(controller.getAndClearCount());
        } catch (InterruptedException e) {
        } // try
      } // while
    } // run()

    // Передает значение счетчика количества машин.
    private native void transmit(int count);
  } // class TrafficTransmitter
```

И наконец, приведем класс `TrafficSensorController`. Экземпляры класса `TrafficSensorController` хранят текущее общее количество машин, прошедших мимо сенсоров движения транспорта, связанных с экземпляром. Заметим, что его методы реализованы как синхронизированные.

```
  public class TrafficSensorController
            implements TrafficSensor.TrafficObserver {
    private int vehicleCount = 0;
...
    /**
     * Этот метод вызывается в том случае, когда сенсор движения
     * транспорта фиксирует прохождение машины. Он увеличивает
     * значение счетчика машин на единицу.
     */
    public synchronized void vehiclePassed() {
      vehicleCount++;
    } // vehiclePassed()

    /**
     * Сбрасывает счетчик машин в нуль.
     * @return Возвращает последнее значение счетчика.
     */
    public synchronized int getAndClearCount() {
      int count = vehicleCount;
      vehicleCount = 0;
      return count;
    } // getAndClearCount()
  } // class TrafficSensorController
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ SINGLE THREADED EXECUTION

Почти все другие шаблоны для конкурирующих операций используют шаблон Single Threaded Execution.

Lock Object (Объект блокировки)

СИНОПСИС

Должна выполняться операция, требующая однопоточного доступа к многочисленным объектам. Чтобы сэкономить время и упростить задачу, при помощи шаблона Lock Object создается дополнительный объект блокировки, который заменяет все объекты, непосредственно участвующие в операции. Объект, который предназначен только для того, чтобы быть заблокированным, называется *объектом блокировки.*

КОНТЕКСТ

Предположим, создается программа для игры, которая будет происходить в реальном времени и в которой будет участвовать множество игроков. Каждый игрок сможет инициировать операции, которые влияют на состояние многочисленных игровых объектов. Чтобы гарантировать надлежащее изменение состояния каждого объекта, при разработке необходимо применить шаблон Single Threaded Execution.

Чтобы игра правильно функционировала, изменения некоторых игровых объектов должны происходить одновременно и не влиять на действия другого игрока. Для этого нужно гарантировать следующее: поток, отвечающий за изменение состояния объектов, должен иметь монопольный доступ к этим объектам.

С этой целью можно потребовать от потоков блокировать каждый объект перед тем, как они пытаются изменить его состояние. Решение такого рода связано с рядом сложностей.

1. Блокирование многочисленных объектов требует дополнительных ресурсов компьютера, но это наименьшая из проблем.
2. Необходимо быть уверенным, что при попытке блокировать несколько объектов потоки не окажутся взаимно блокированными. Не должна возникнуть ситуация, при которой один поток блокирует объект, который уже заблокирован другим потоком, и этот объект не будет освобожден до тех пор, пока второй поток не сможет получить доступ к другому объекту, заблокированному первым потоком.

   Подобных взаимных блокировок можно избежать, если использовать шаблон Static Locking Order, описанный в книге [Grand2001]. В таком случае заставляют потоки сначала сортировать объекты в соответствии с тем значением, которое возвращает для них `System.identityHashCode`, и затем блокируют их в этом порядке.

Очевидно, что попытка избежать взаимной блокировки усложняет программу и требует дополнительных затрат.

3. Блокирование коллекции объектов требует больших трудозатрат при кодировании на языке Java. Чтобы заблокировать произвольное количество объектов, необходимо выполнять манипуляции внутри конструктора `synchronized` рекурсивного метода. Рассмотрим пример:

```
public class ObjectLocker {
   /**
    * Этот метод блокирует все объекты данного массива
    * и передает массив объекту ObjectManipulationIF.
    *
    * @param objs
    *     Массив объектов, на который воздействует
    *     данный объект ObjectManipulationIF.
    * @param op
    *     Объект ObjectManipulationIF.
    */
   public void doIt(Object[] objs, ObjectManipulationIF op) {
      doItHelper(objs, op, objs.length-1);
   } // doIt(Object[], ObjectManipulationIF)

   private void doItHelper(Object[] objs,
                           ObjectManipulationIF op,
                           int ndx) {
      synchronized (objs[ndx]){
         if (ndx==0) {
            op.doObject(objs);
         } else {
            doItHelper(objs, op, ndx-1);
         } // if
      } // synchronized
   } // doItHelper(Object[], ObjectManipulationIF, int)
} // class ObjectLocker
```

Очевидно, что желательно найти альтернативное решение этой проблемы: обеспечить для потока монопольный доступ к многочисленным игровым объектам. При этом решение не должно быть таким сложным и требующим больших затрат, как решение, предусматривающее блокировку многочисленных объектов.

Проблему можно упростить, если заставить только один поток в какой-то момент времени получать доступ к любым объектам игры. Это ограничение может

вынудить потоки ждать, хотя в противном случае они не должны были бы это делать. Однако меньшая сложность и меньшие затраты того стоят.

Для реализации этой упрощенной стратегии создают один новый объект, который будет использоваться только как объект блокировки. Правило таково, что ни один поток не сможет каким-либо образом повлиять на состояние игрового объекта до тех пор, пока он не заблокирует объект блокировки.

МОТИВЫ

☺ Чтобы обеспечить правильность выполнения операций, должны выполняться операции, требующие монопольного доступа к объектам, на которые эти операции воздействуют.

☺ Необходимость блокировки произвольного набора объектов с целью выполнения некоторой операции усложняет эту операцию и требует дополнительных затрат на нее.

☺ Вполне допустимо, чтобы перед тем, как начаться, операция иногда ожидала завершения другой операции. При этом требования к производительности являются настолько умеренными, что иногда приходится ждать завершения другой операции даже в том случае, когда в этом нет необходимости.

РЕШЕНИЕ

Нужно организовать потоки таким образом, чтобы они получали монопольный доступ к набору объектов, выполняя блокировку синхронизации объекта, который существует только для этой цели. Взаимодействие, предусматривающее использование объекта блокировки, описывается шаблоном, представленным на рис. 9.6.

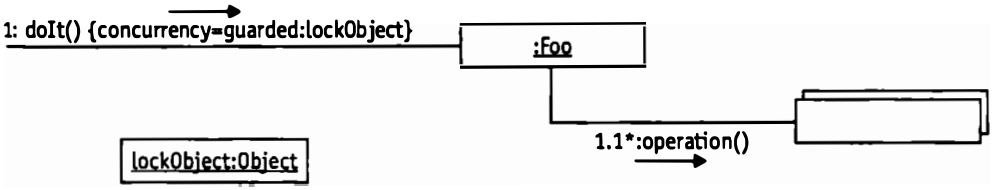


Рис. 9.6. Взаимодействие классов, использующих объект блокировки

Как показано на схеме, вызывается метод `doIt` объекта `Foo`. Метод `doIt` начинает с блокировки объекта `lockObject`. После этого он выполняет операции с другими объектами, а затем снимает блокировку объекта блокировки.

Объект блокировки может быть внедрен в некоторую структуру классов различными способами. Например, на рис. 9.7 изображен абстрактный класс, который имеет статический метод `getLockObject`. Подклассы этого абстрактного

класса вызывают метод `getLockObject` с целью получения объекта блокировки для синхронизированных операций. Такой способ управления объектом блокировки хорошо работает в тех случаях, когда нужно, чтобы все экземпляры одного или нескольких классов совместно использовали один и тот же объект блокировки.

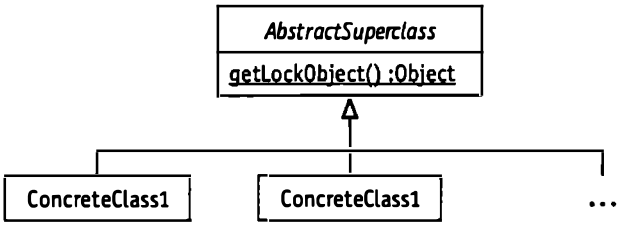


Рис. 9.7. Объект блокировки

Другие способы управления объектами блокировки рассматриваются в разделе «Реализация». Поскольку существует множество приемлемых способов управления доступом к объектам блокировки, они на самом деле не являются частью данного шаблона. Этот шаблон посвящен использованию объекта блокировки, а не управлению доступом к нему.

РЕАЛИЗАЦИЯ

Существует множество различных способов управления доступом к объекту блокировки. Основным доводом в пользу того или иного способа управления объектом блокировки является разнообразие тех объектов, которые будут использовать объект блокировки, а также ответ на вопрос, будет нужен один или несколько объектов блокировки.

Пример управления объектом блокировки, представленный на рис. 9.7, предусматривает использование классов, которые получат доступ к объекту блокировки через наследуемый ими статический метод. Такая схема хорошо работает в тех случаях, когда нужно заблокировать все экземпляры класса или классов для выполнения некоторой операции.

Если почти все объекты, блокируемые для выполнения операции, не являются экземплярами одного и того же класса, их, как правило, можно сгруппировать по другому принципу. В таких случаях способ управления объектом блокировки обычно определяется на основе схемы объектов, участвующих в операциях, для которых предназначен объект блокировки. В качестве примера рассмотрим структуру объектов, имеющую вид дерева (рис. 9.8).

Объекты, организованные в виде дерева, являются экземплярами разных классов. Но все объекты имеют метод `getLockObject`, который возвращает объект блокировки, используемый операциями, выполняемыми с участием этих объектов. Метод `getLockObject` реализован в этих объектах таким образом, что,

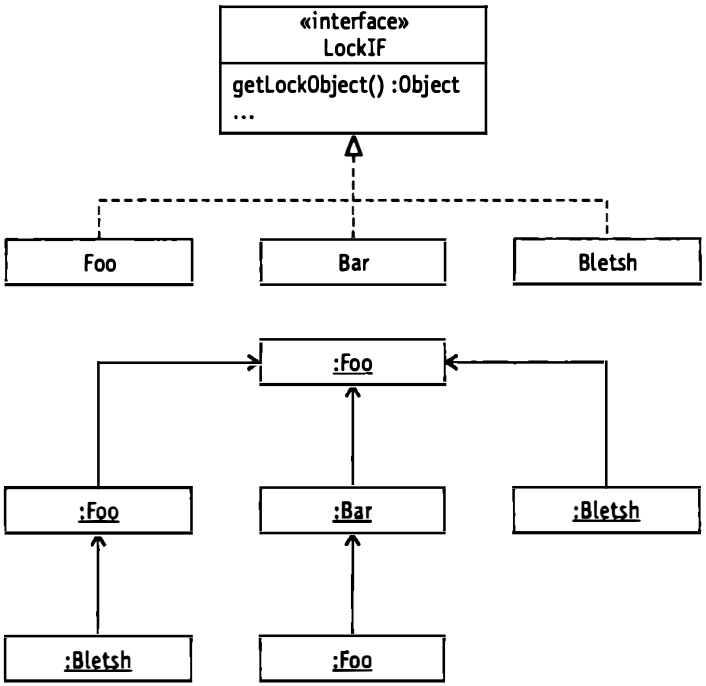


Рис. 9.8. Объекты дерева блокировки

если объекты не являются корнями дерева, их метод `getLockObject` вызывает метод `getLockObject` родительского объекта. Если объект является корнем дерева, он предоставляет реальный объект блокировки.

Если операция нуждается в монопольном доступе к любому из объектов, образующих дерево, она произвольным образом выбирает один из объектов, доступ к которому должен быть монопольным, и вызывает его метод `getLockObject`. Метод `getLockObject` любого из объектов дерева возвращает один и тот же объект блокировки. Как только операция заблокирует объект блокировки, она получает монопольный доступ ко всем объектам дерева.

СЛЕДСТВИЯ

☺ Применяя шаблон Lock Object, можно гарантировать, что в какой-то момент времени доступ к набору объектов получает только один поток. Это реализуется достаточно просто и не требует больших издержек.

☹ Операция, использующая шаблон Lock Object, получает монопольный доступ ко всем объектам, которые могут принимать участие в этой операции или любой другой операции. Это означает, что, если две операции могли бы выполняться параллельно, поскольку они не влияют на один и тот же объект, они не смогут этого сделать.

ПРИМЕНЕНИЕ В JAVA API

Класс `java.awt.Component` имеет метод `getTreeLock`. Этот метод возвращает объект, который используется классом `Component` и всеми его подклассами как объект блокировки. Метод `getTreeLock` каждого объекта `Component` возвращает один и тот же объект блокировки.

Подклассы класса `Component` используются для создания GUI, основанных на AWT (Abstract Windows Toolkit, пакет инструментальных средств для абстрактных окон[1]) и Swing.

ПРИМЕР КОДА

Пример кода для этого шаблона включает некоторые классы, которые предусматриваются сценарием использования игровых объектов, описанным в разделе «Контекст». Ниже приводится часть абстрактного суперкласса тех классов, которые применяются для моделирования классов игры.

```
public abstract class AbstractGameObject {
   private static final Object lockObject = new Object();
   ...
   /**
    * True, если этот объект выделяется.
    */
   private boolean glowing;

   ...

   public static final Object getLockObject() {
      return lockObject;
   } // getLockObject()

   ...

   public boolean isGlowing() {
      return glowing;
   } // isGlowing()

   public void setGlowing(boolean newValue) {
      glowing = newValue;
   } // setGlowing(boolean)
} // class AbstractGameObject
```

[1] Иногда говорят просто: «оконная библиотека Java». (*Примеч. ред.*)

Обратите внимание, что класс AbstractGameObject имеет метод getLockObject. Этот метод возвращает объект, который будет использоваться как объект блокировки. Кроме того, этот класс имеет свойство типа boolean, позволяющее определять, выделяется ли объект. Следующий листинг представляет собой код для класса AbstractGameObject, который использует вышеупомянутые метод и свойство.

```
class GameCharacter extends AbstractGameObject {
   ...
   private ArrayList myWeapons = new ArrayList();
   public void dropAllWeapons() {
      synchronized (getLockObject()) {
         for (int i = myWeapons.size()-1; i>=0; i-) {
            ((Weapon)myWeapons.get(i)).setGlowing(true);
         } // for
         ...
      } // synchronized
   } // dropAllWeapons()

   ...
} // class GameCharacter
```

Класс GameCharacter имеет метод dropAllWeapons, который начинает блокировку объекта блокировки, возвращенного методом getLockObject. Он наследует этот метод от класса AbstractGameObject. Затем он устанавливает в true свойство свечения объекта оружия, связанного с этим героем. Обладая монопольным доступом ко всем игровым объектам, он продолжает делать все остальные необходимые действия.

ШАБЛОН ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЙ С ШАБЛОНОМ LOCK OBJECT

Single Threaded Execution. Шаблон Lock Object представляет собой более изящный вариант шаблона Single Threaded Execution.

Guarded Suspension (Охраняемая приостановка)

Этот шаблон основан на материале, который был представлен в работе [Lea97].

СИНОПСИС

Если существует условие, которое запрещает методу выполнять его функции, то данный шаблон приостанавливает выполнение этого метода до того момента, пока условие не перестанет действовать.

КОНТЕКСТ

Предположим, нужно создать класс, который реализует структуру очереди (queue) данных. Это структура данных вида «первым зашел — первым вышел», т.е. объекты удаляются из очереди в том же порядке, в котором они добавлялись в очередь. На рис. 9.9 представлен класс `Queue`.

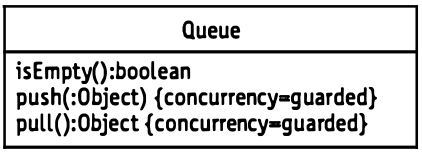


Рис. 9.9. Класс Queue

Класс `Queue` имеет два интересных для нас метода:

- `push` добавляет объекты в очередь;
- `pull` удаляет объекты из очереди.

Если метод `pull` объекта `Queue` вызывается в тот момент, когда очередь пуста, он не возвратит объект из очереди до тех пор, пока в очереди не появится объект, который должен будет возвращаться методом `pull`. Объект может быть добавлен в очередь в то время, пока метод `pull` находится в состоянии ожидания, если другой поток вызовет метод `push`. Эти два метода синхронизированы, что позволяет многочисленным потокам безопасно выполнять параллельные обращения к ним.

Если просто сделать оба метода синхронизированными, может возникнуть следующая проблема: в момент вызова метода `pull` объекта `Queue` очередь пуста. Метод `pull` будет ожидать, пока в результате вызова метода `push` не появится объект, который можно будет возвратить. Но поскольку оба метода синхронизированы, обращения к методу `push` не смогут выполниться до тех пор, пока не

закончит работу метод `pull`, а метод `pull` не закончит выполнение до тех пор, пока не выполнится метод `push`.

Решением этой проблемы является задание предварительного условия для метода `pull`, чтобы он не выполнялся, если очередь пуста.

На рис. 9.10 показаны параллельные обращения к методам `push` и `pull` объекта `Queue`. Если метод `pull` вызывается в тот момент, когда метод `isEmpty` объекта `Queue` возвращает true, то поток ожидает до тех пор, пока метод `isEmpty` не возвратит false. Только тогда сможет выполниться метод `pull`. На самом деле невозможно выполнить метод `pull` в то время, пока очередь пуста, поэтому не существует проблемы с вызовом метода `push`, который может добавлять объекты в пустую очередь.

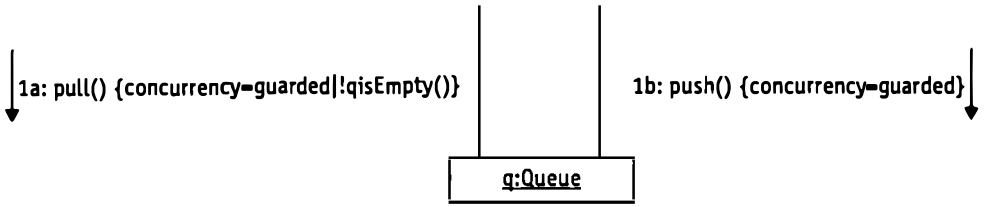


Рис. 9.10. Взаимодействие для очереди

МОТИВЫ

☺ Чтобы иметь возможность выполнять безопасные параллельные обращения к методам класса, эти методы должны быть синхронизированными.

☺ Объект может находиться в таком состоянии, когда один из его синхронизированных методов не может быть выполнен полностью. Чтобы вывести объект из такого состояния, должен быть выполнен другой синхронизированный метод этого объекта. Если первому методу разрешить выполнение в то время, пока объект находится в этом состоянии, произойдет взаимная блокировка и он никогда не завершится. Обращения к методу, изменяющему состояние объекта и дающему возможность первому методу завершиться, будут ожидать завершения первого метода, чего никогда не произойдет.

РЕШЕНИЕ

Рассмотрим рис. 9.11, на котором показан класс элемента управления окном `Widget`. Этот класс имеет два синхронизированных метода: `foo` и `bar`. Объекты `Widget` могут иметь особое состояние. Когда объект `Widget` находится в таком состоянии, его метод `isOK` возвращает false; в противном случае он возвращает true. Если объект `Widget` находится в особом состоянии, обращение к его методу `bar` может вывести его из этого состояния. Помимо вызова метода `bar`, не

существует другого способа вывести объект Widget из особого состояния. Выведение объекта Widget из особого состояния — это побочный эффект выполнения основной функции метода bar, поэтому нельзя вызывать метод bar только для того, чтобы вывести из особого состояния объект Widget.

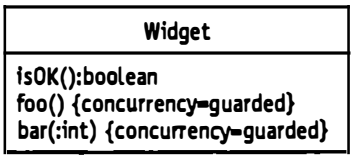


Рис. 9.11. Класс Widget

Обращение к методу foo объекта Widget не может завершиться, если объект Widget находится в особом состоянии. Если это случилось, то, поскольку методы foo и bar являются синхронизированными, последующие обращения к методам foo и bar объекта Widget не будут выполнены до тех пор, пока не закончится выполнение предыдущего вызова метода foo. Вызов метода foo не закончит выполнение до тех пор, пока вызов метода bar не выведет объект Widget из особого состояния.

Назначение шаблона Guarded Suspension состоит в том, чтобы не допустить ситуацию взаимной блокировки, когда поток должен выполнить синхронизированный метод объекта, но состояние этого объекта не позволяет методу закончить выполнение. Если вызов метода происходит в тот момент, когда состояние объекта препятствует завершению выполнения метода, шаблон Guarded Suspension приостанавливает поток до тех пор, пока состояние объекта не позволит методу завершиться (рис. 9.12).

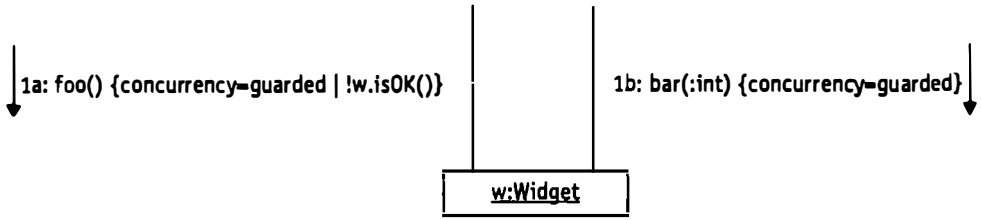


Рис. 9.12. Взаимодействие для шаблона Guarded Suspension

Заметим, что на рис. 9.12 указано предварительное условие, которое должно быть удовлетворено перед тем, как начнет выполняться обращение к методу foo объекта Widget. Если поток пытается вызвать метод foo объекта Widget в тот момент, когда метод isOK объекта Widget возвращает false, поток должен будет ожидать до тех пор, пока метод isOK не возвратит true; тогда сможет выполниться метод foo. Пока такой поток ждет возврата значения true от метода isOK, другие потоки могут беспрепятственно вызывать метод bar.

РЕАЛИЗАЦИЯ

Шаблон Guarded Suspension реализуется с использованием методов `wait` и `notify` следующим образом:

```
class Widget {
   synchronized void foo(){
      while (!isOK()) {
         wait();
      }
      ...
   }

   synchronized void bar(x int) {
      ...
      notify();
   }
}
```

Этот механизм действует так: метод, например `foo`, перед тем как начать реально выполняться, должен проверить выполнение некоторого предусловия. Прежде всего этот метод должен проверить это предусловие в цикле `while`. Если предусловие ложно, то вызывается метод `wait`.

Каждый класс наследует метод `wait` от класса `Object`. Когда поток вызывает метод `wait` объекта, этот метод заставляет поток заблокировать объект. Затем метод ждет, пока ему не будет сообщено о разрешении разблокировать объект. Далее, как только поток сможет снова осуществлять блокировку, метод `wait` заканчивает свою работу.

После возвращения метода `wait` управление снова передается в самое начало цикла `while`, где повторяется проверка предварительного условия. Проверка предварительных условий в цикле обусловлена тем, что в промежутке между тем моментом времени, когда поток пытается в первый раз осуществить синхронизационную блокировку, и тем моментом времени, когда ему это удается, другой поток может сделать предварительные условия ложными.

Вызов метода `wait` оповещается о необходимости выполнения в том случае, когда другой метод, например, `bar`, вызывает метод `notify` объекта. Методы вызывают метод `notify` после того, как они изменили состояние объекта таким образом, чтобы могли удовлетворяться предварительные условия метода. Метод `notify` представляет собой другой метод, наследуемый всеми классами от класса `Object`. Задача метода `notify` в том, чтобы оповестить другой поток, ожидающий окончания выполнения метода `wait`, о необходимости окончания метода `wait`.

Если в состоянии ожидания находится несколько потоков, метод `notify` выбирает один из них случайным образом. Произвольный выбор хорошо работает

в большинстве ситуаций. Но он не сможет работать нормально для таких объектов, которые имеют методы с разными предварительными условиями. Рассмотрим ситуацию, когда несколько обращений к методам ожидают удовлетворения своих разных предварительных условий. Произвольный выбор может привести к тому, что будут удовлетворены предварительные условия одного метода, но поток, который получил извещение, ожидает выполнения метода, имеющего другие предварительные условия, которые еще не выполнены. Тогда существует вероятность того, что метод никогда не закончит завершение, так как этот метод никогда не будет извещен об удовлетворении своих предварительных условий.

Для тех классов, которые не приемлют произвольного выбора при принятии решения, существует альтернативный способ решения вопроса, какой поток известить. Методы таких классов могут вызвать метод `notifyAll`. Вместо оповещения одного выборочного потока метод `notifyAll` оповещает все ожидающие потоки. При этом решается проблема, связанная с отсутствием оповещения нужного потока, но могут расходоваться лишние системные ресурсы ввиду активизации потоков, ожидающих выполнения обращений к методам, предварительные условия которых не выполнены.

СЛЕДСТВИЯ

☺ Шаблон Guarded Suspension позволяет потоку решать проблемы объекта, состояние которого не позволяет выполнить некоторую операцию, посредством приостановки до того момента, пока объект не будет в состоянии выполнить эту операцию.

- Шаблон Guarded Suspension использует синхронизированные методы или синхронизированные команды. Возможна такая ситуация, когда несколько потоков ждут выполнения вызова одного и того же метода. Шаблон Guarded Suspension специально не занимается вопросами выбора, какому из ожидающих потоков будет разрешено продолжить выполнение, если состояние объекта позволит выполнить метод. Для этой цели можно использовать шаблон Scheduler.

☹ Если существует вложенная синхронизация, использование шаблона Guarded Suspension может быть весьма затруднительным. Рассмотрим диаграмму взаимодействия (рис. 9.13).

Доступ к объекту `Widget` осуществляется при помощи синхронизированных методов объекта `Thing`. Это значит, что если поток `1a` вызывает метод `foo` объекта `Widget` именно в тот момент, когда состояние объекта `Widget` заставляет его метод `isOK` возвращать false, то поток будет бесконечно ждать, пока метод `isOK` объекта `Widget` не возвратит true. Причина в том, что методы объекта `Thing` синхронизированы без каких-либо предварительных условий. Возникшая проблема является именно той задачей, для решения которой и предназначен шаблон Guarded Suspension.

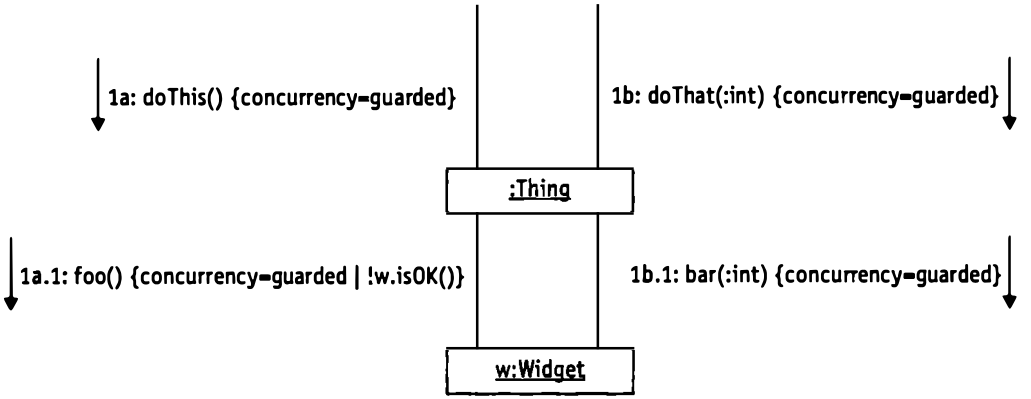


Рис. 9.13. Guarded Suspension в условиях вложенной синхронизации

ПРИМЕНЕНИЕ В JAVA API

Класс `java.net.InetAddress` использует шаблон Guarded Suspension с целью управления внутренней оптимизацией. Одной из обязанностей класса `InetAddress` является нахождение в сети адреса, который соответствует имени хоста. Поскольку для определения адреса, соответствующего некоторому имени в сети, нужно обращаться к одному или нескольким удаленным серверам, эта операция является достаточно длительной.

Как только класс `InetAddress` определил сетевой адрес, соответствующий имени хоста, он помещает этот адрес в кэш. Кэш является общим для всех объектов `InetAddress`. Когда объект `InetAddress` должен находить в сети адрес, соответствующий имени хоста, прежде всего он ищет его в кэше. Если искомый адрес содержится в кэше, объекту `InetAddress` не нужно обращение к какому-либо удаленному серверу.

В то время когда объект `InetAddress` получает адрес, соответствующий некоторому имени хоста, другому объекту `InetAddress` может понадобиться адрес, соответствующий тому же имени хоста. Пока первый объект добывает этот адрес, нет никакого смысла в том, чтобы второй объект искал тот же адрес в кэше. Адрес не появится в кэше до тех пор, пока первый объект `InetAddress` не получит его. Самое лучшее, что может сделать второй объект `InetAddress`, — это ждать, пока первый объект `InetAddress` не найдет этот адрес. Именно в такой ситуации класс `InetAddress` использует шаблон Guarded Suspension.

Перед тем как проверить, не содержит ли кэш адрес хоста, объект `InetAddress` должен узнать, не занят ли другой объект `InetAddress` получением этого же адреса. Если другой объект `InetAddress` получает искомый адрес, то первый объект `InetAddress` ожидает до тех пор, пока второй объект `InetAddress` не получит этот адрес и не положит его в кэш. Только тогда первый объект `InetAddress` продолжит свои действия и обратится в кэш с целью получения нужного адреса хоста.

ПРИМЕР КОДА

Приведем код, реализующий проект класса `Queue`, рассмотренный в разделе «Контекст»:

```
public class Queue {
   private ArrayList data = new ArrayList();

   /**
    * Помещаем объект в конец очереди.
    */
   synchronized public void put(Object obj) {
      data.add(obj);
      notify();
   } // put(Object)

   /**
    * Получаем объект, находящийся в начале очереди.
    * Если очередь пуста, ждем, пока в ней не появится объект.
    */
   synchronized public Object get() {
      while (data.size() == 0){
         try {
            wait();
         } catch (InterruptedException e) {
         } // try
      } // while
      Object obj = data.get(0);
      data.remove(0);
      return obj;
   } // get()
} // class Queue
```

Обратите внимание, что в последнем листинге вызов метода `wait` находится внутри оператора `try`, который генерирует исключительную ситуацию `InterruptedException`. При вызове метода `wait` может быть сгенерирована эта исключительная ситуация. Самое простое поведение программы — проигнорировать `InterruptedException`, которая может генерироваться методом `wait`. В описании шаблона Two-Phase Termination содержатся пояснения, когда метод `wait` генерирует `InterruptedException` и что нужно делать в этом случае.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ GUARDED SUSPENSION

Balking. Шаблон Balking предоставляет другую стратегию обработки обращений к методам объектов, состояние которых не позволяет выполнить эти методы.

Two-Phase Termination. Реализация шаблона Two-Phase Termination обычно предполагает генерацию и обработку исключительной ситуации `InterruptedException`. В его реализации, как правило, присутствует взаимодействие с шаблоном Guarded Suspension.

Balking (Отмена)

Этот шаблон основан на материале, представленном в работе [Lea97].

СИНОПСИС

Если метод объекта вызывается в тот момент, когда состояние объекта не позволяет его выполнить, метод заканчивает выполнение, ничего не сделав.

КОНТЕКСТ

Предположим, создается программа, управляющая электронным устройством смывания для туалета. Эти устройства предназначены для использования в общественных уборных. Смывное устройство имеет датчик освещенности, установленный на его передней панели. Когда он фиксирует повышение уровня освещенности, он полагает, что человек вышел из туалета, и приводит в движение струю воды. Электронное смывное устройство для туалета имеет также кнопку, которая может использоваться для ручного включения водяной струи. На рис. 9.14 изображена диаграмма классов, моделирующих такое поведение.

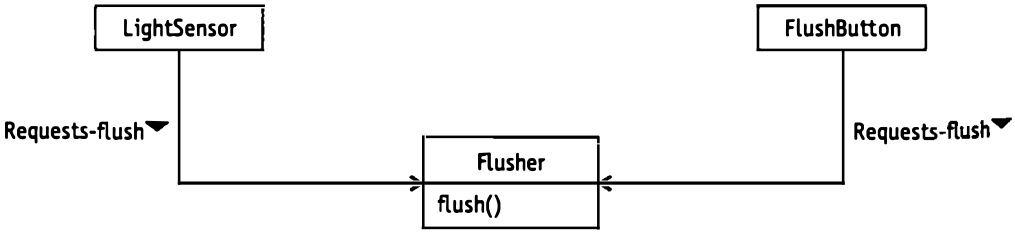


Рис. 9.14. Классы смывного устройства

Если объект `LightSensor` или объект `FlushButton` решает, что нужна струя воды, для ее запуска он запрашивает объект `Flusher`, вызывая его метод `flush`. Метод `flush` запускает струю воды и, как только появляется водяной поток, завершается. Такой механизм подразумевает необходимость решения некоторых вопросов конкурентности.

Нужно решить, что произойдет в том случае, если в момент вызова метода `flush` водяная струя уже приведена в действие. Придется решить также, что будет, когда оба объекта, `LightSensor` и `FlushButton`, вызывают метод `flush` объекта `Flusher` одновременно.

Опишем наиболее очевидные варианты управления обращением к методу `flush` в тот момент, когда водяная струя уже приведена в действие.

1. Немедленный запуск новой водяной струи. Запуск новой струи в тот момент, когда одна водяная струя уже приведена в действие, приводит к такому же эффекту, что и более продолжительное по сравнению с нормальным действие уже существующей водяной струи. Оптимальная длительность нормальной струи должна быть определена экспериментально. Более продолжительная водяная струя может привести к излишнему расходу воды, поэтому этот вариант не очень хорош.
2. Ждать, пока не закончится одна водяная струя, а затем немедленно запустить другую. Такой вариант на самом деле вдвое увеличивает продолжительность водяной струи, поэтому предполагает еще более значительный расход воды, чем первый вариант.
3. Ничего не делать. Этот вариант не предусматривает излишнего расхода воды, поэтому он является наилучшим.

Если существуют два параллельных обращения к методу `flush`, то удачной стратегией считается разрешение выполнения одного вызова и игнорирование другого.

Предположим, что обращение к методу объекта производится в тот момент, когда состояние объекта не позволяет ему правильно выполнить метод. Если управление ситуацией таково, что метод завершается, не выполнив свою задачу, значит, была произведена *отмена метода*. В UML нет стандартного способа, позволяющего обозначить метод, вызов которого сопровождается его отменой. В данной книге вызов метода, сопровождающегося отменой, обозначается в виде стрелки, изогнутой в обратном направлении.

На рис. 9.15 представлено такое поведение метода `flush` класса `Flusher`.

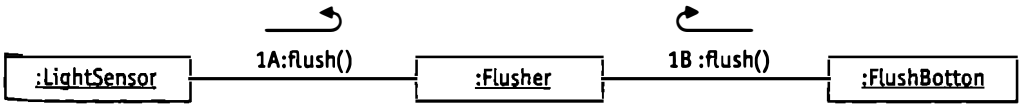


Рис. 9.15. Взаимодействие для класса Flusher

МОТИВЫ

☺ Объект может находиться в состоянии, которое не позволяет выполнить обращение к его методу.

☺ Задержка выполнения вызова метода до тех пор, пока состояние объекта не станет подходящим, — это неприемлемая политика при решении данной проблемы. Правильный результат может быть получен только в том случае, если метод выполнится немедленно.

☺ Обращения к методу объекта, сделанные в тот момент, когда состояние объекта не позволяет выполнить метод, могут быть проигнорированы без опасных последствий.

РЕШЕНИЕ

Диаграмма взаимодействия, представленная на рис. 9.16, включает объекты, сотрудничающие в рамках шаблона Balking.

Объект `Client` вызывает метод `doIt` объекта `Service`. Изогнутая в обратном направлении стрелка указывает на то, что может быть произведена отмена метода. Если метод `doIt` объекта `Service` вызывается в тот момент, когда состояние объекта `Service` не позволяет выполнить обращение к его методу `doIt`, то этот метод возвращается, не выполнив свои обычные функции.

Метод `doIt` возвращает результат, обозначенный на диаграмме как `didIt`. Этот результат может иметь значение true или false в зависимости от того, выполнил ли он свои обычные функции или была произведена отмена метода.

Рис. 9.16. Взаимодействие, демонстрирующее отмену метода

РЕАЛИЗАЦИЯ

Если можно выполнить отмену метода, то прежде всего он, как правило, проверяет состояние объекта, которому он принадлежит, с целью определения, должен ли он быть выполнен.

Когда метод объекта выполняется, решив, что не будет отменен, недопустимо, чтобы состояние объекта стало вдруг неприемлемым для выполнения этого метода. Чтобы зашититься от подобной несогласованности, можно использовать шаблон Single Threaded Execution.

Вместо того чтобы сообщить о своей отмене всем, кто его вызывал, посредством возврата некоторого значения, метод может также известить их о том, что была не произведена отмена, посредством генерации исключительной ситуации. Если вызывающую сторону не интересует, была ли произведена отмена метода, он может не возвращать такую информацию.

СЛЕДСТВИЯ

☺ Обращения к методам не будут выполнены, если они производятся в тот момент, когда состояние объектов не позволяет это сделать.

- Обращение к методу, который можно было отменить, означает, что метод может не выполнить свои обычные функции и при этом вообще ничего не сделать.

ПРИМЕР КОДА

Приведем код для класса `Flusher`, рассмотренного в разделе «Контекст»:

```
public class Flusher {
   private boolean flushInProgress = false;

   /**
    * Этот метод вызывается для запуска водяной струи.
    */
   public void flush() {
      synchronized (this) {
         if (flushInProgress)
            return;
         flushInProgress = true;
      }
      // Здесь должен быть код для активизации водяной струи.
      ...
   }

   /**
    * Этот метод вызывается с той целью, чтобы известить
    * данный объект о завершении действия водяной струи.
   */
   void flushCompleted() {
      flushInProgress = false;
   } // flushCompleted()
} // class Flusher
```

Если метод `flush` вызывается в тот момент, когда водяная струя уже действует, будет произведена отмена этого метода.

Обратите внимание, что в методе `flush` использован оператор синхронизации. Его присутствие объясняется тем, что, если многочисленные обращения к методу `flush` происходят одновременно, результат будет ожидаемым. Только один вызов будет выполняться нормально, а другие будут отменены. Если бы не было оператора синхронизации, многочисленные обращения к методу могли бы выполняться нормально в один и тот же момент времени.

Также следует заметить, что метод `flushCompleted` не синхронизирован. Это объясняется тем, что присваивание значения false переменной `flushInProgress` никогда не приведет к появлению проблемы. Поскольку метод `flushCompleted` не изменяет никакую другую информацию о состоянии, параллельные обращения к методу `flushCompleted` вполне безопасны. Кроме

того, не может быть параллельных обращений к методу `flushCompleted`, поскольку устройство механизма смывного устройства таково, что оно физически не способно активизировать два водяных потока одновременно.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ BALKING

Guarded Suspension. Шаблон Guarded Suspension предоставляет альтернативный способ управления обращениями к методам объектов, состояние которых не позволяет выполнить эти обращения.

Single Threaded Execution. Шаблон Balking часто используется вместе с шаблоном Single Threaded Execution для согласования изменений состояния объекта.

Scheduler (Планировщик)

Этот шаблон основан на материале, представленном в работе [Lea97].

СИНОПСИС

Управляет порядком, в соответствии с которым потокам предстоит выполнить последовательный код, используя для этой цели объект, который явным образом задает последовательность ожидающих потоков. Шаблон Scheduler обеспечивает механизм реализации политики планирования, но при этом не зависит ни от одной конкретной политики.

КОНТЕКСТ

Предположим, проектируется ПО для управления физической безопасностью зданий. Система безопасности должна включать контрольные пункты. Прежде чем пройти через такой пункт, человек должен провести идентификационной карточкой через сканер. При этом контрольный пункт либо разрешает человеку пройти, либо нет. В любом случае: прошел ли человек через контрольный пункт, или его карточка не была принята — входные данные будут распечатаны в журнале регистрации, находящемся в центральном офисе службы безопасности.

Взаимодействия на рис. 9.17 включают объекты `SecurityCheckpoint`, которые создают объекты `JournalEntry` и передают их методу `print` объекта `Printer`. Несмотря на простоту диаграммы, с этой структурой связана одна проблема. Она возникает в том случае, когда люди проходят через три или более пропускных пункта примерно в одно и то же время. Когда принтер распечатывает первую запись регистрации, другие обращения к принтеру находятся в состоянии ожидания. После окончания распечатки первой регистрационной записи неизвестно, какая запись будет распечатана следующей. Это означает, что записи регистрации могут распечатываться не в том порядке, в котором контрольные пункты безопасности передавали их на принтер.

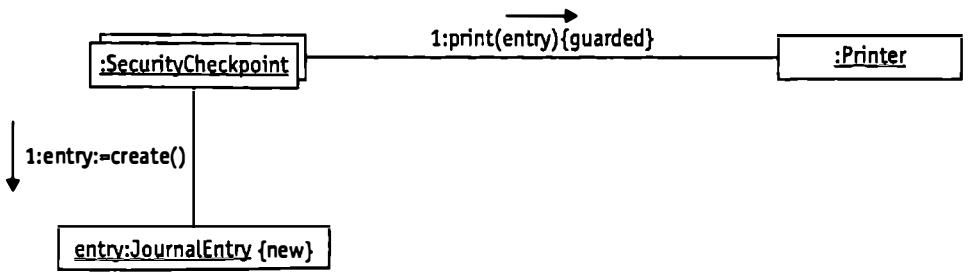


Рис. 9.17. Взаимодействие для журнала системы безопасности

Чтобы записи в журнале распечатывались в порядке их фактического поступления, можно просто помещать каждую журнальную запись в очередь и затем распечатывать журнальные записи в том порядке, в котором они заносились в очередь. Хотя здесь все еще возможно одновременное поступление трех или более журнальных записей, вероятность такой ситуации значительно снижается. Для распечатки журнальной записи может потребоваться одна секунда. Чтобы возникла проблема, необходимо, чтобы две другие журнальные записи поступили в течение этого промежутка времени. Помещение журнальной записи в очередь может потребовать всего лишь примерно одной микросекунды. При этом вероятность нарушения порядка распечатки журнальных записей уменьшается в миллион раз.

Класс `Printer` мог бы отвечать за организацию очереди журнальных записей, подлежащих выводу на печать. Однако помещение в очередь обращений к методам, подлежащих последовательному выполнению, — это характеристика, которая обладает большим потенциалом с точки зрения многократного использования, если она реализуется как отдельный класс. Представленная на рис. 9.18 диаграмма взаимодействия демонстрирует, как объект принтера может взаимодействовать с другими объектами, организуя очередь на выполнение обращений к его методу `print`.

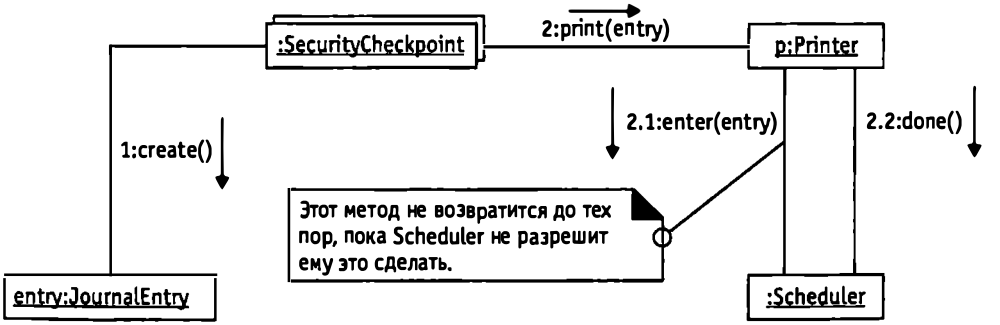


Рис. 9.18. Журнал системы безопасности, использующий объект-планировщик

Объект `SecurityCheckpoint` вызывает метод `print` объекта `printer`. Метод `print` начинается с обращения к методу `enter` объекта `Scheduler`. Метод `enter` не заканчивает свою работу до тех пор, пока объект `Scheduler` не разрешит ему это сделать. Когда метод `print` заканчивает свою работу, он вызывает метод `done` объекта `Scheduler`. В промежутке времени между возвратом метода `enter` и вызовом метода `done` объект `Scheduler` полагает, что управляемый им ресурс занят. Ни одно из обращений к его методу `enter` не будет выполнено, пока объект `Scheduler` считает, что управляемый им ресурс занят. Это гарантирует, что в некоторый момент времени только один поток выполняет ту часть метода `print`, которая начинается с его обращения к методу `enter` объекта `Scheduler` и заканчивается его обращением к методу `done` объекта `Scheduler`.

Реальная логика выполнения, согласно которой объект `Scheduler` принимает решение, когда должен выполниться вызов метода `enter`, инкапсулирована в объекте `Scheduler`. Это позволяет менять логику, не затрагивая другие объекты. В данном примере, когда несколько обращений к методу `enter` ожидают выполнения, можно применить следующую логику управления.

- Если объект `Scheduler` не ожидает обращения к своему методу `done`, то вызов его метода `enter` будет выполнен немедленно. Затем объект `Scheduler` будет ожидать вызова своего метода `done`.
- Если объект `Scheduler` ожидает обращения к своему методу `done`, то вызов его метода `enter` будет ждать своего выполнения до тех пор, пока не произойдет вызов метода `done` объекта `Scheduler`. Если при вызове метода `done` объекта `Scheduler` имеются какие-либо обращения к его методу `enter`, ожидающие выполнения, то для выполнения будет выбрано одно из таких обращений к методу `enter`.
- Если своего выполнения ожидают несколько обращений к методу `enter` объекта `Scheduler`, то объект `Scheduler` должен выбрать следующий вызов метода `enter`, которому будет разрешено выполниться. Он выберет тот, который был передан объекту `JournalEntry` раньше всего. Если несколько объектов `JournalEntry` имеют одинаковое время поступления, то один из них будет выбран произвольно.

Чтобы класс `Scheduler` мог сравнивать время создания объектов `JournalEntry` и оставаться при этом независимым от класса `JournalEntry`, он не должен прямо ссылаться на класс `JournalEntry`. Он может ссылаться на класс `JournalEntry` через интерфейс (рис. 9.19).

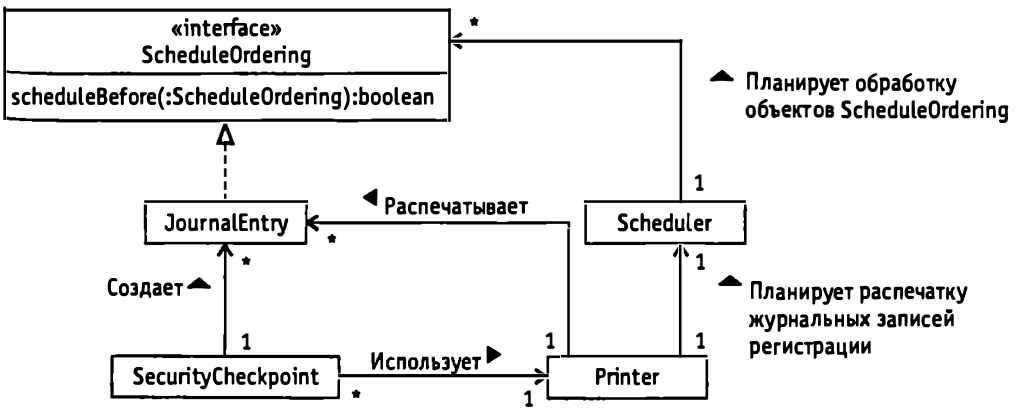


Рис. 9.19. Классы планирования регистрационных записей в журнале

Класс `Scheduler` ничего не знает о классе `JournalEntry`. Он просто планирует обработку объектов, реализующих интерфейс `ScheduleOrdering`. Этот интерфейс объявляет метод `scheduleBefore`, который вызывается классом

Scheduler с целью определения, какой из двух объектов ScheduleOrdering должен быть обработан первым. Хотя класс Scheduler инкапсулирует политику управления, позволяющую определять, когда объекту ScheduleOrdering будет дано разрешение на обработку, он делегирует принятие решения, касающееся последовательности обработки, объекту ScheduleOrdering.

МОТИВЫ

☺ Несколько потоков могут потребовать доступа к ресурсу одновременно, и только один поток в какой-то момент времени может осуществить доступ к ресурсу.

☺ Согласуясь с требованиями программы, потоки должны осуществлять доступ к ресурсу в определенном порядке.

РЕШЕНИЕ

Шаблон Scheduler использует некоторый объект, чтобы явным образом управлять параллельными запросами с целью их непараллельной обработки. На рис. 9.20 показаны классы и интерфейс в шаблоне Scheduler.

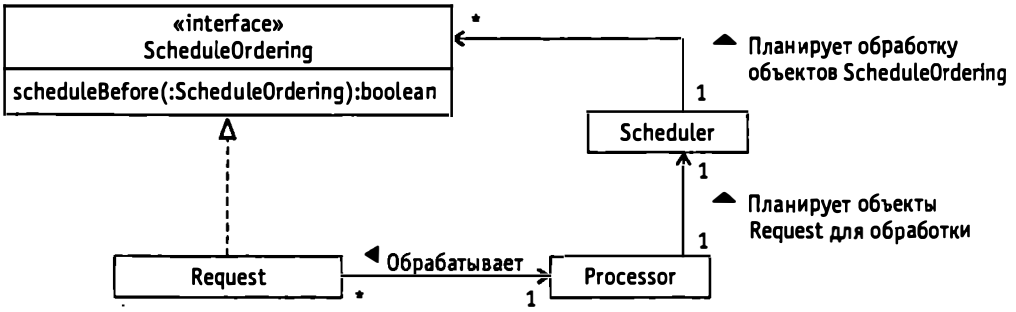


Рис. 9.20. Классы шаблона Scheduler

Опишем роли, исполняемые этими классами и интерфейсом.

Request. Выступающие в этой роли классы должны реализовывать интерфейс, исполняющий роль ScheduleOrdering. Объекты Request инкапсулируют запрос к объекту Processor на выполнение каких-либо действий.

Processor. Экземпляры классов в этой роли выполняют вычисление, описанное в объекте Request. Экземпляры классов могут быть представлены несколькими объектами Request, но они могут выполнить только одну операцию, представленную объектом Request, в некоторый момент времени. Объект Processor делегирует объекту Scheduler ответственность за управление обработкой объектов Request. В какой-то момент должен обрабатываться только один объект.

Scheduler. Экземпляры классов в этой роли управляют обработкой объектов `Request`, выполняемой объектом `Processor`. Чтобы быть независимыми от типов запросов, класс `Scheduler` не должен ничего знать об управляемом им классе `Request`. Вместо этого он осуществляет доступ к объектам `Request` через реализуемый ими интерфейс `ScheduleOrdering`.

Класс в этой роли отвечает за принятие решения о том, когда будет выполнен следующий запрос. Он не отвечает за порядок выполнения запросов. Он делегирует эту ответственность интерфейсу `ScheduleOrdering`.

ScheduleOrdering. Объекты `Request` реализуют интерфейс, который выполняет эту роль. Выступающие в этой роли интерфейсы предназначены для решения задач двух типов.

1. Ссылаясь на интерфейс `ScheduleOrdering`, классы `Processor` не зависят от класса `Request`.
2. Вызывая методы, определяемые интерфейсом `ScheduleOrdering`, классы `Scheduler` способны делегировать принятие решения о том, какой объект `Request` будет обработан следующим. Тем самым достигается независимость от типов решаемых задач классами `Scheduler`. На рис. 9.20 такой метод — `scheduleBefore`.

Взаимодействие между объектом `Processor` и объектом `Scheduler` происходит в два этапа (рис. 9.21). Оно начинается с обращения к методу `doIt` объекта `Processor`. Метод `doIt` вызывает метод `enter` объекта `Scheduler`, связанного с объектом `Processor`. Если в данный момент никакой другой поток не выполняет остальную часть метода `doIt`, то метод `enter` выполняется немедленно. После выполнения метода `enter` объект `Scheduler` знает, что управляемый им ресурс занят. Когда ресурс занят, не будет выполнен ни один вызов метода `enter` объекта `Scheduler` до тех пор, пока ресурс не освободится и объект `Scheduler` не придет к решению, что подошла очередь выполнить очередной запрос.

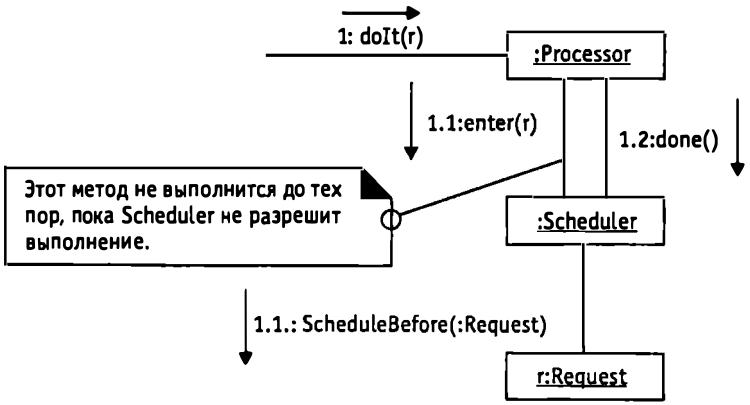


Рис. 9.21. Взаимодействие для Scheduler

Объект Scheduler считает управляемый им ресурс занятым до тех пор, пока не будет вызван метод done объекта Scheduler. Когда один поток обращается к методу done объекта Scheduler и какие-либо другие потоки ожидают выполнения метода enter объекта Scheduler, тогда один из них будет выбран для выполнения.

Если вызов метода enter объекта Scheduler должен ожидать своего выполнения и имеются другие вызовы, ожидающие выполнения метода enter, то объект Scheduler должен решить, какой вызов будет выполнен на этот раз. Принятие решения он делегирует объектам Request, которые были переданы этим методам в качестве параметров с целью определения, какой вызов будет выполнен следующим. Объект Scheduler делает это через вызов методов, объявленных интерфейсом SchedulеOrdering специально для этой цели и реализованных объектом Request.

РЕАЛИЗАЦИЯ

В некоторых случаях использования шаблона Scheduler класс Scheduler реализует такую политику планирования, которая не требует от него консультаций с объектами Request с целью определения порядка выполнения обращений к его методу enter. Примером такой политики может служить поведение, при котором разрешенный порядок выполнения обращений к методам enter соответствует порядку их вызова. В таких случаях нет необходимости передавать объекты Request методу enter или создавать интерфейс SchedulеOrdering. Другим примером этой политики может служить поведение, при котором порядок планирования запросов не имеет значения, но ставится условие, определяющее интервал между концом одного задания и началом другого, например, равный или превышающий пять минут.

СЛЕДСТВИЯ

☺ Шаблон Scheduler предоставляет способ явного контроля в такой ситуации, когда потоки могут выполнять код.

☺ Политика планирования инкапсулирована в своем собственном классе и является многократно используемой.

☹ Использование шаблона Scheduler может служить причиной значительных дополнительных издержек, помимо расходов, необходимых для простого вызова синхронизированного метода.

ПРИМЕР КОДА

Приведем код, реализующий проект планирования вывода на печать, рассмотренный в разделе «Контекст». В первом листинге — реализация класса Printer, управляющего распечаткой записей журнала контрольного пункта безопасности.

```
class Printer {
   private Scheduler scheduler = new Scheduler();

   public void print(JournalEntry j) {
      try {
         scheduler.enter(j);
         try {
            ...
         } finally {
            scheduler.done();
         } // try
      } catch (InterruptedException e) {
      } // try
} // class Printer
```

Каждый объект `Printer` использует объект `Scheduler` для планирования параллельных обращений к его методу `print`, поэтому они распечатываются последовательно и в соответствии с их временем поступления. Сначала вызывается метод `enter` объекта `Scheduler`, которому передается распечатываемый объект `JournalEntry`. Вызов не выполнится до тех пор, пока объект `Scheduler` не решит, что подошла очередь распечатать этот объект `JournalEntry`.

Метод `print` заканчивается обращением к методу `done` объекта `Scheduler`. Вызов метода `done` говорит объекту `Scheduler` о том, что объект `JournalEntry` распечатан, и может подойти очередь вывода на печать другого объекта `JournalEntry`.

Приведем исходный код для класса `Scheduler`:

```
public class Scheduler {
   private Thread runningThread;
```

Переменная `runningThread` устанавливается в `null`, если управляемый объектом `Scheduler` ресурс не занят. Если ресурс занят, она содержит ссылку на поток, использующий этот ресурс.

```
   private ArrayList waitingRequests = new ArrayList();
   private ArrayList waitingThreads = new ArrayList();
```

Инвариантом для этого класса является то, что запрос и соответствующий ему поток находятся только в `waitingRequests` и `waitingThread`, пока обращение к методу `enter` ожидает выполнения. Запросы, ожидающие выполнения, и соответствующие потоки содержатся в разных объектах `ArrayList`, что способствует принятию решения, какой запрос должен быть обработан следующим.

Метод `enter` вызывается перед тем, как поток начнет использовать управляемый ресурс. Метод `enter` не выполняется до тех пор, пока управляемый ресурс

не освободится и объект Scheduler не примет решение, что подошла очередь выполнения этого запроса.

```
public void enter(ScheduleOrdering s)
              throws InterruptedException {
     Thread thisThread = Thread.currentThread();

   // Когда управляемый ресурс свободен, выполняется синхронизация
   // этого объекта, чтобы два параллельных вызова
   // метода enter не были выполнены одновременно.
   synchronized (this) {
      if (runningThread == null) {
          runningThread = thisThread;
          return;
      } // if
      waitingThreads.add(thisThread);
      waitingRequests.add(s);
   } // synchronized (this)
   synchronized (thisThread) {
      // Ожидает, пока вызов метода done другим потоком не решит,
      // что пришла очередь этого потока.
      while (thisThread != runningThread) {
         thisThread.wait();
      } // while
   } // synchronized (thisThread)
   synchronized (this) {
      int i = waitingThreads.indexOf(thisThread);
      waitingThreads.remove(i);
      waitingRequests.remove(i);
   } // synchronized (this)
} // enter(ScheduleOrdering)
```

Вызов метода done указывает на то, что текущий поток завершил работу, и управляемый ресурс освободился.

```
synchronized public void done() {
     if (runningThread != Thread.currentThread())
        throw new IllegalStateException("Wrong Thread");
     int waitCount = waitingThreads.size();
     if (waitCount <= 0) {
        runningThread = null;
```

```
      } else if (waitCount == 1) {
         runningThread = (Thread)waitingThreads.get(0);
         waitingThreads.remove(0);
      } else {
         int next = waitCount - 1;
         ScheduleOrdering nextRequest;
         nextRequest
            = (ScheduleOrdering)waitingRequests.get(next);
         for (int i = waitCount-2; i>=0; i-) {
            ScheduleOrdering r;
            r = (ScheduleOrdering)waitingRequests.get(i);
            if (r.scheduleBefore(nextRequest)) {
               next = i;
               nextRequest = (ScheduleOrdering)
                                          waitingRequests.get(next);
            } // if
         } // for
         runningThread = (Thread)waitingThreads.get(next);
         synchronized (runningThread) {
            runningThread.notifyAll();
         } // synchronized (runningThread)
      } // if waitCount
   } // done()
} // class Scheduler
```

Для активизации потока метод done использует метод notifyAll, а не метод notify, так как нельзя с уверенностью утверждать, что не существует другого потока, также ожидающего получения права на блокировку объекта runningThread. Если используется метод notify и существуют другие потоки, также ожидающие получения права на блокировку объекта runningThread, то метод notify может активизировать не тот поток, который нужен.

Класс Scheduler не зависит от конкретного типа управляемого ресурса. Но он требует, чтобы классы, описывающие операции над управляемым ресурсом, реализовывали интерфейс ScheduleOrdering. Если несколько операций ожидают доступа к ресурсу, класс Scheduler использует интерфейс ScheduleOrdering для определения порядка выполнения операций. Листинг интерфейса ScheduleOrdering:

```
public interface ScheduleOrdering {
   public boolean scheduleBefore(ScheduleOrdering s);
} // interface ScheduleOrdering
```

И наконец, примерный код класса `JournalEntry`, который должен быть распечатан классом `Printer`:

```
public class JournalEntry implements ScheduleOrdering {
   ...
   private Date time = new Date();
...

   /**
    * Возвращает время создания этого JournalEntry.
    */
   public Date getTime() { return time; }

   /**
    * Возвращает true, если данный запрос должен
    * обрабатываться перед этим запросом.
    */
   public boolean scheduleBefore(ScheduleOrdering s) {
      if (s instanceof JournalEntry)
         return getTime().before(((JournalEntry)s).getTime());
      return false;
   } // scheduleBefore(ScheduleOrdering)
} // class JournalEntry
```

ШАБЛОН ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЙ С ШАБЛОНОМ SCHEDULER

Read/Write Lock. Некоторые реализации шаблона Read/Write Lock обычно используют шаблон Scheduler с целью осуществления качественного планирования.

Read/Write Lock (Блокировка чтения/записи)

Этот шаблон основан на материале, представленном в работе [Lea97].

СИНОПСИС

Разрешает объекту осуществлять параллельный доступ для операций чтения, но осуществляет монопольный доступ для операций записи.

КОНТЕКСТ

Предположим, что разрабатывается ПО для проведения интерактивных аукционов. На аукцион будут выставляться предметы. Пользователи будут подключаться к интерактивному аукциону для просмотра текущей цены на предмет, а затем принимать решение, хотят ли они указать бóльшую цену за предмет. В какое-то заранее определенное время аукцион закроется, и тот, кто предложил на данный момент наивысшую цену, получит предмет по окончательной предложенной цене.

Ожидается, что на чтение текущей цены предмета поступит намного больше запросов, чем на ее изменение. Для координирования доступа к предложениям можно использовать шаблон Single Threaded Execution. Хотя он и гарантирует правильность результатов, но может значительно снизить способность к реагированию. Если несколько пользователей хотят прочитать текущую цену одновременно, шаблон Single Threaded Execution разрешает в какой-то момент прочитать текущее предложение цены только одному пользователю. При этом пользователи, которые хотят только прочитать текущее предложение цен, непременно должны ждать других пользователей, которые тоже просто хотят прочитать текущее предложение.

Не существует причин, препятствующих множеству пользóвателей читать текущее предложение цен одновременно. Последовательное выполнение необходимо только для изменения текущей цены. В какой-то момент должно производиться только одно изменение текущей цены, чтобы другие изменения, не повышающие текушее значение, были отброшены.

Шаблон Read/Write Lock позволяет избежать длительного ожидания при чтении данных, разрешая параллельные операции чтения, но допуская только последовательный доступ к данным при их обновлении.

На рис. 9.22 представлено несколько объектов пользовательского интерфейса, вызывающих методы `getBid` и `setBid` объекта предложенной цены. Перед тем как возвратить текушую предложенную цену, метод `getBid` ожидает до тех пор, пока не будет ни одного, ожидающего завершения, обращения к методу

setBid. Перед тем как изменить текущую предложенную цену, метод setBid ожидает завершения выполнения всех обращений к методам getBid или setBid. С целью возможности повторного использования объект ReadWriteLock инкапсулирует логику, координирующую выполнение методов getBid и setBid.

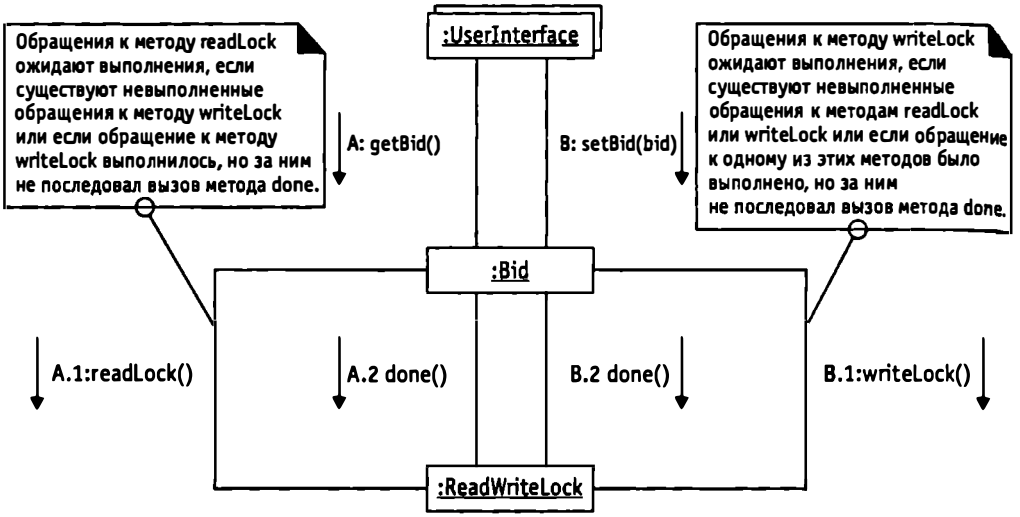


Рис. 9.22. Взаимодействие для предложения цены на аукционе

Все обращения к методу readLock объекта ReadWriteLock выполняются немедленно, если нет каких-либо невыполненных блокировок записи. Блокировка записи считается невыполненной в течение некоторого интервала времени, который начинается с момента возврата метода writeLock объекта ReadWriteLock и заканчивается моментом его освобождения соответствующим вызовом метода done объекта ReadWriteLock. Обращения к методу readLock ожидают выполнения до тех пор, пока не будут освобождены все невыполненные блокировки записи.

Обращения к методу writeLock объекта ReadWriteLock выполняются немедленно, если не является истинным хотя бы одно из следующих утверждений:

- ожидает выполнения предыдущий вызов метода writeLock;
- предыдущее обращение к методу writeLock завершилось, но не было соответствующего вызова метода done объекта ReadWriteLock;
- имеются какие-либо выполняющиеся обращения к методу readLock объекта ReadWriteLock или невыполненные блокировки чтения.

Если обращение к объекту ReadWriteLock производится в тот момент, когда какое-либо из вышеуказанных условий является истинным, оно не выполняется до тех пор, пока все указанные условия не станут false.

МОТИВЫ

☺ Существует необходимость доступа для чтения и записи информации о состоянии объекта.

☺ Считывание информации о состоянии объекта может параллельно выполняться любым количеством операций. Однако операции чтения гарантируют возврат правильного значения только в том случае, если одновременно с операцией чтения не выполняются операции записи.

☺ В какой-то момент времени должна выполняться только одна операция записи информации о состоянии объекта, это гарантирует правильность выполнения этой операции.

☺ Предполагается наличие параллельно инициируемых операций чтения.

☺ При разрешении параллельного выполнения параллельно активизируемых операций чтения повышается быстрота реагирования и производительность.

☺ Логика, предназначенная для согласования операций чтения и записи, должна быть повторно используемой.

РЕШЕНИЕ

Нужно организовать класс таким образом, что параллельные обращения к методам, считывающим и сохраняющим информацию из его экземпляра, координируются экземпляром другого класса. На рис. 9.23 показаны роли, исполняемые классами в шаблоне Read/Write Lock.

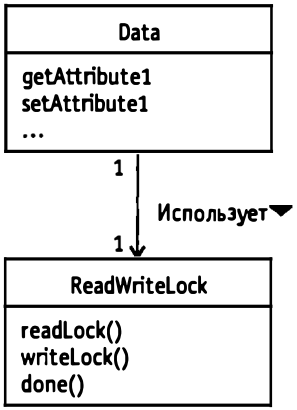


Рис. 9.23. Классы в шаблоне Read/Write Lock

Класс в роли `Data` имеет методы для чтения и записи информации своего экземпляра. Любому количеству потоков разрешено параллельно считывать информацию из экземпляра объекта `Data` до тех пор, пока не появится поток, од-

новременно записывающий информацию. С другой стороны, только одна операция записи информации должна выполняться в тот момент времени, когда не выполняются операции считывания. Объекты `Data` должны согласовывать свои операции записи и чтения таким образом, чтобы данные ограничения неукоснительно выполнялись.

Блокировка чтения — это абстракция, которая используется объектами `Data` для координации операций чтения. Методы чтения объекта `Data` не производят выборку какой-либо информации до тех пор, пока они не выполнят блокировку чтения. С каждым объектом `Data` связан объект `ReadWriteLock`. Перед тем как один из его методов `get` начнет что-нибудь считывать, он вызывает метод `readLock` объекта `ReadWriteLock`, который осуществляет блокировку чтения для текущего потока. Если поток заблокирован, метод `get` может быть уверен в безопасном считывании данных объекта. Это объясняется тем, что объект `ReadWriteLock` не будет осуществлять блокировку записи, пока существуют какие-либо невыполненные блокировки чтения. Если при вызове метода `readLock` объекта `ReadWriteLock` имеются какие-либо невыполненные блокировки записи, этот метод не будет выполнен до тех пор, пока все невыполненные блокировки записи не будут освобождены посредством обращений к методу `done` объекта `ReadWriteLock`. Во всех остальных случаях обращения к методу `readLock` объекта `ReadWriteLock` выполняются немедленно.

Когда метод `get` объекта `Data` закончит считывание данных объекта, он вызывает метод `done` объекта `ReadWriteLock`. Обращение к этому методу заставляет текущий поток снять блокировку чтения.

Таким же образом объекты `Data` используют абстракцию блокировки записи с целью согласования операций чтения. Методы `set` объекта `Data` не сохраняют информацию до тех пор, пока не осуществят блокировку записи. Перед тем как один из методов `set` объекта `Data` запишет какую-либо информацию, он вызывает метод `writeLock` соответствующего объекта `ReadWriteLock`, который выполняет для текущего потока блокировку записи. Пока поток обладает блокировкой записи, метод записи может быть уверен в безопасном задании данных объекта. Это объясняется тем, что объект `ReadWriteLock` осуществляет блокировку записи только в том случае, когда нет невыполненных блокировок чтения и записи. Если в момент вызова метода `writeLock` объекта `ReadWriteLock` имеются какие-либо невыполненные блокировки, он не будет выполнен до тех пор, пока все невыполненные блокировки не будут освобождены посредством обращения к методу `done` объекта `ReadWriteLock`.

Вышеописанные ограничения, которые управляют блокировкой чтения и записи, не касаются порядка самого блокирования. Порядок блокировки чтения не имеет значения до тех пор, пока операции чтения могут выполняться параллельно. Поскольку операции записи выполняются по очереди, порядок блокирования записи должен соответствовать порядку выдачи запросов на блокировку.

Существует некоторая неопределенность в том случае, когда имеются обращения сразу к двум методам, `readLock` и `writeLock`, объекта `ReadWriteLock`,

ожидающие выполнения, и нет невыполненных блокировок. Если операции чтения должны возвращать самую последнюю информацию, то первым должен выполняться метод `writeLock`.

РЕАЛИЗАЦИЯ

Блокировка чтения и блокировка записи не содержат никакой информации, поэтому их не нужно представлять в виде конкретных объектов. Достаточно просто подсчитать их.

Разные варианты политики планирования, представленные в данном описании шаблона Read/Write Lock, отдают предпочтение блокировке чтения. Это означает, что если некоторый вызов ожидает получения блокировки записи, поскольку существуют невыполненные блокировки чтения, то любые запросы, желающие получить дополнительные блокировки чтения, в таких обстоятельствах будут удовлетворены немедленно. Подобная политика хороша для многих приложений. Но иногда используются и другие приложения, где более уместна такая политика планирования, которая отдает предпочтение блокировкам записи.

Если некоторая политика планирования отдает предпочтение блокировкам записи, это означает то, что никакие блокировки чтения не будут инициированы до тех пор, пока существует вызов, ожидающий активизации блокировки записи. Возможны и другие варианты поведения.

СЛЕДСТВИЯ

☺ Шаблон Read/Write Lock координирует параллельные обращения к методам `get` и `set` объекта таким образом, чтобы обращения к методам `set` не мешали ни друг другу, ни обращениям к методам `get`.

☺ При наличии большого количества параллельных обращений к методам `get` объекта шаблон Read/Write Lock, применяемый с целью согласования этих обращений к методам `get`, может продемонстрировать в конечном счете лучшую способность к реагированию и производительность, чем используемый для той же цели шаблон Single Threaded Execution. Объясняется это тем, что шаблон Read/Write Lock разрешает параллельное выполнение параллельных обращений к методам `get` объекта.

• При наличии относительно небольшого количества параллельных обращений к методам `get` объекта применение шаблона Read/Write Lock приведет в результате к более низкой производительности, чем использование Single Threaded Execution. Это объясняется тем, что шаблон Read/Write Lock затрачивает больше времени на управление отдельными обращениями.

☹ Если политика планирования отдает предпочтение блокировкам чтения, то существует вероятность того, что блокировки записи никогда не будут инициированы. Если существует такой достаточно постоянный поток запросов

на блокировку чтения, что всегда имеется по крайней мере одна невыполненная блокировка чтения, то в таком случае ни одна блокировка записи никогда не сможет быть инициирована. Эта ситуация называется *зависанием записи* (write starvation).

☹ Аналогичным образом, если политика планирования отдает предпочтение блокировкам чтения, тогда возможна такая ситуация, когда блокировки чтения никогда не будут инициированы. Если существует такой достаточно постоянный поток запросов блокировки записи, что всегда имеется по крайней мере одна невыполненная блокировка записи, в таком случае ни одна блокировка чтения никогда не сможет быть инициирована. Эта ситуация называется *зависанием чтения* (read starvation).

Существуют подходы, которые позволяют избежать проблем с ситуациями такого рода. Они состоят в том, чтобы изменить предпочтение на противоположное, если число ожидающих запросов превысило заранее определенный лимит. Другой способ устранения заключается в том, чтобы задавать предпочтение случайным образом каждый раз, когда должен производиться выбор определенной политики планирования. Недостатком таких подходов является то, что они усложняют поведение класса `ReadWriteLock`, затрудняя его анализ и отладку.

ПРИМЕР КОДА

Приведем код, реализующий проект, который был рассмотрен в разделе «Контекст». Сначала — листинг достаточно простого класса `Bid`.

```
public class Bid {
   private int bid = 0;
   private ReadWriteLock lockManager = new ReadWriteLock();
...
   public int getBid() throws InterruptedException{
      lockManager.readLock();
      int bid = this.bid;
      lockManager.done();
      return bid;
   } // getBid()

   public void setBid(int bid) throws InterruptedException {
      lockManager.writeLock();
      if (bid > this.bid) {
         this.bid = bid;
      } // if
      lockManager.done ();
   } // setBid(int)
} // class Bid
```

Очевидно, что методы класса Bid используют объект ReadWriteLock для согласования параллельных обращений. Сначала (перед чтением или записью значений) они вызывают соответствующий метод блокировки. По завершении они вызывают метод done объекта ReadWriteLock с целью разблокировки.

Класс ReadWriteLock более сложен. По мере прочтения его листинга можно заметить, что он уделяет основное внимание двум моментам.

1. Тщательно отслеживает информацию о состоянии, используя при этом механизм, который совместим со всеми потоками.
2. Обеспечивает выполнение всех предварительных условий, прежде чем выполнятся его методы блокировки.

Любые другие классы, отвечающие за проведение политики планирования, должны будут позаботиться о реализации этих идей.

```
public class ReadWriteLock {
   private int waitingForReadLock = 0;
   private int outstandingReadLocks = 0;
   private ArrayList waitingForWriteLock = new ArrayList();
   private Thread writeLockedThread;
```

Объект ReadWriteLock использует эти переменные экземпляра для отслеживания потоков, которые запросили или получили запрос на блокировку чтения или записи. Для отслеживания потоков, которые ожидают блокировки записи, он использует список, на который ссылается переменная waitingForWriteLock. Применение такого списка позволяет осуществлять блокировки записи в том порядке, в котором выдавались соответствующие запросы.

Объект ReadWriteLock использует переменную waitingForReadLock для подсчета количества потоков, ожидающих блокировки чтения. Простой подсчет является вполне достаточным, поскольку все потоки, ожидающие блокировки чтения, получат разрешение на них одновременно. Это значит, что не нужно отслеживать порядок запрашивания потоками блокировки чтения.

Объект ReadWriteLock использует переменную outstandingReadLocks для подсчета количества блокировок чтения, которые были инициированы, но еще не освобождены соответствующими потоками, для которых они предназначались.

Объект ReadWriteLock использует переменную writeLockedThread для ссылки на поток, у которого в данный момент блокирована запись. Если в данный момент времени ни у одного потока не блокирована запись, то переменная writeLockedThread содержит null. Имея переменную, ссылающуюся на поток, у которого была блокирована запись, объект ReadWriteLock знает, был ли поток активизирован с целью блокирования записи или по другой причине.

Метод readLock объекта ReadWriteLock действует следующим образом. Он инициирует блокировку чтения и немедленно заканчивает выполнение, если

нет невыполненной блокировки записи. Блокировка чтения состоит в том, что значение в переменной `outstandingReadLocks` увеличивается на единицу.

```
synchronized public void readLock() throws InterruptedException {
     if (writeLockedThread != null) {
        waitingForReadLock++;
        while (writeLockedThread != null) {
           wait();
        } // while
        waitingForReadLock-;
     } // if
     outstandingReadLocks++;
  } // readLock()
```

Дальше — листинг метода `writeLock`. Как можно заметить, этот листинг больше, чем листинг метода `readLock`. Это объясняется тем, что `writeLock` управляет потоками и структурой данных. Код начинается с того, что проверяется случай отсутствия невыполненных блокировок. Если нет невыполненных блокировок, немедленно осуществляется блокировка записи. В противном случае текущий поток добавляется в список, который рассматривается методом `done` как очередь. Текущий поток ожидает до тех пор, пока метод `done` не разрешит ему блокировку записи. Затем метод `writeLock` завершается, удаляя текущий поток из списка потоков, ожидающих блокировку записи.

```
public void writeLock() throws InterruptedException {
     Thread thisThread;
     synchronized (this) {
        if ( writeLockedThread==null
              && outstandingReadLocks==0) {
           writeLockedThread = Thread.currentThread();
           return;
        } // if
        thisThread = Thread.currentThread();
        waitingForWriteLock.add(thisThread);
     } // synchronized(this)
  synchronized (thisThread) {
     while (thisThread != writeLockedThread) {
           thisThread.wait();
        } // while
     } // synchronized (thisThread)
     synchronized (this) {
           waitingForWriteLock.remove (thisThread);
     } // synchronized (this)
  } // writeLock
```

Метод done завершает листинг класса ReadWriteLock. Потоки вызывают метод done объекта ReadWriteLock для снятия блокировки, ранее произведенной объектом ReadWriteLock. Метод done рассматривает три случая.

1. Существуют невыполненные блокировки чтения, наличие которых предполагает, что нет невыполненных блокировок записи. Он снимает блокировку чтения посредством уменьшения на единицу переменной outstandingReadLock. Если больше нет невыполненных блокировок чтения и существуют потоки, ожидающие блокировки записи, то блокируется запись для того потока, который дольше всех ждет этой блокировки. Затем ожидающий поток активизируется.
2. Существует невыполненная блокировка записи. Метод done заставляет текущий поток снять блокировку записи. Если имеются какие-либо потоки, ожидающие блокировки чтения, он инициирует блокировки чтения для всех этих потоков. Если нет невыполненных блокировок чтения и есть потоки, ожидающие блокировки записи, этот метод блокирует запись того потока, который ожидает дольше всех, заставляя переменную writeLockedThread ссылаться на этот, а не на текущий поток.
3. Нет невыполненных блокировок. Если нет невыполненных блокировок, это значит, что метод done был вызван ошибочно, поэтому он генерирует исключение IllegalStateException.

```
synchronized public void done() {
     if (outstandingReadLocks > 0) {
        outstandingReadLocks-;
        if ( outstandingReadLocks==0
           && waitingForWriteLock.size()>0) {
           writeLockedThread
              = (Thread)waitingForWriteLock.get(0) ;
           writeLockedThread.notifyAll();
        } // if
     } else if (Thread.currentThread()
              == writeLockedThread) {
        if ( outstandingReadLocks==0
           && waitingForWriteLock.size()>0) {
           writeLockedThread
              = (Thread)waitingForWriteLock.get(0) ;
           writeLockedThread.notifyAll();
        } else {
           writeLockedThread = null;
           if (waitingForReadLock > 0)
              notifyAll();
        } // if
```

```
        } else {
            String msg = "Thread does not have lock";
            throw new IllegalStateException(msg);
        } // if
    } // done()
} // class ReadWriteLock
```

Отметим еще одну последнюю деталь в отношении метода `done`: вместо метода `notify` он использует метод `notifyAll`. Если ему нужно разрешить блокировку чтения, он вызывает метод `notifyAll` объекта `ReadWriteLock`, чтобы позволить продолжить выполнение всем потокам, ожидающим блокировки чтения. Когда он осуществляет для некоторого потока блокировку записи, он вызывает метод `notifyAll` этого потока. Вызов метода `notify` будет работать в большинстве случаев. Однако в том случае, когда другой поток ожидает блокировки синхронизированного потока с целью блокировки записи, использование метода `notify` может привести к активизации неправильного потока. Применение метода `notifyAll` гарантирует активизацию нужного потока.

Существует более мощная версия класса `ReadWriteLock`, которая является частью ПО ClickBlocks, которое можно найти на сайте www.clickblocks.org. Этот класс находится в пакете под названием `org.clickblocks.util`.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ READ/WRITE LOCK

Single Threaded Execution. Шаблон Single Threaded Execution — это хорошая и более простая альтернатива шаблону Read/Write Lock в том случае, когда большая часть доступа к данным представляет собой доступ для записи.

Scheduler. Шаблон Read/Write Lock — это специальная форма шаблона Scheduler.

Producer-Consumer (Производитель-потребитель)

СИНОПСИС

Координирует асинхронное производство и использование информации или объектов.

КОНТЕКСТ

Предположим, что проектируется система диспетчеризации претензий, которые будут поступать от потребителей через интернет. Диспетчеры будут просматривать претензии и определять, куда направлять их для решения.

Любое количество клиентов может в любое время предъявить претензии. На дежурстве обычно находится множество диспетчеров. Если кто-либо из них не занят, система немедленно передаст претензии одному из них. В противном случае претензии будут помещены в очередь, пока их не передадут на просмотр диспетчеру (рис. 9.24).

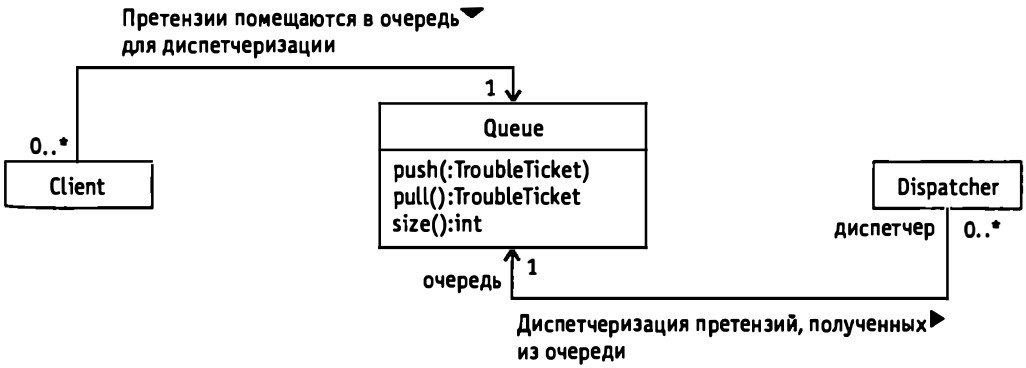


Рис. 9.24. Классы обработки претензий

Экземпляры класса `Client` отвечают за получение форм с претензиями, заполненных пользователями и помещенных в объект `Queue`. Претензии остаются в объекте `Queue` до тех пор, пока объект `Dispatcher` не достанет их из объекта `Queue`.

Класс `Dispatcher` отвечает за получение претензий диспетчером и затем направляет их по адресу назначения, выбранному диспетчером. Когда экземпляр класса `Dispatcher` не передает претензии или не направляет их по назначению,

он вызывает метод `pull` объекта `Queue` с целью получения других претензий. Если в объекте `Queue` нет претензий, метод `pull` ожидает до тех пор, пока не появятся претензии, которые он должен возвратить (рис. 9.25).

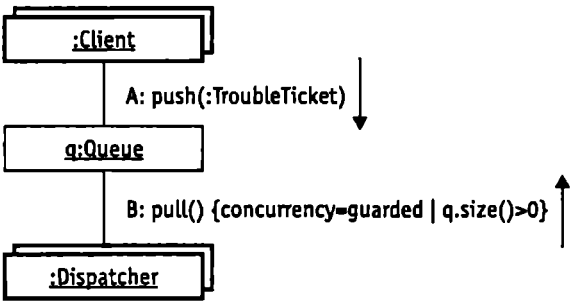


Рис. 9.25. Взаимодействие процесса обработки претензий

МОТИВЫ

☺ Объекты создаются или получаются асинхронно по отношению к их использованию или потреблению.

☺ При создании или получении объекта может случиться так, что не будет какого-либо доступного объекта, который сможет его использовать или потреблять.

РЕШЕНИЕ

Представленная на рис. 9.26 диаграмма описывает классы в рамках шаблона Producer-Consumer.

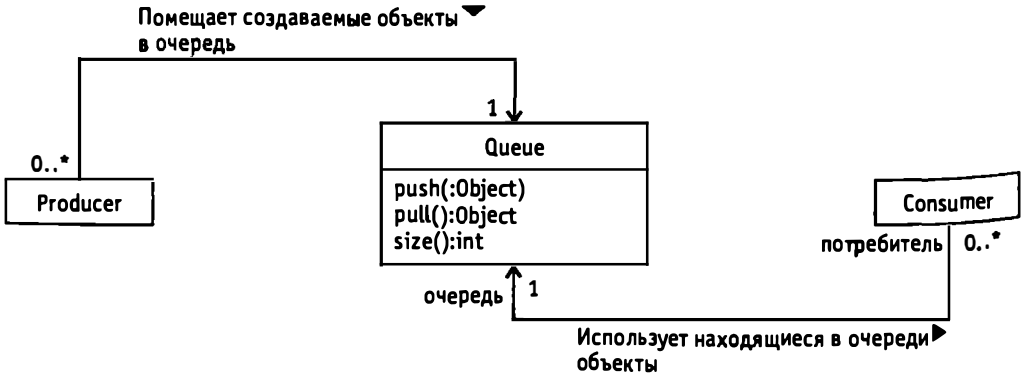


Рис. 9.26. Классы шаблона Producer-Consumer

Опишем роли, которые могут исполняться этими классами.

Producer. Экземпляры классов в этой роли предоставляют объекты, использующиеся объектами `Consumer`. Экземпляры класса `Producer` создают объекты асинхронно по отношению к тем потокам, которые их используют. Это значит, что иногда объект `Producer` может создать объект в тот момент, когда все объекты `Consumer` заняты обработкой других объектов. Экземпляры классов `Producer` не ждут, пока объект `Consumer` станет доступным, вместо этого они помещают созданные ими объекты в очередь и продолжают свою работу.

Queue. Экземпляры классов в этой роли служат в качестве буфера для объектов, созданных экземплярами классов `Producer`. Экземпляры классов `Producer` помещают созданные ими объекты в экземпляр класса `Queue`. Объекты остаются там до тех пор, пока объект `Consumer` не достанет их из объекта `Queue`.

Consumer. Экземпляры классов `Consumer` используют объекты, созданные объектами `Producer`. Они получают используемые ими объекты от объекта `Queue`. Если тот пуст, объект `Consumer`, желающий получить от него объект, ожидает до тех пор, пока объект `Producer` не поместит объект в объект `Queue`.

Представленная на рис. 9.27 диаграмма взаимодействия демонстрирует взаимодействия между объектами, принимающими участие в шаблоне Producer-Consumer.

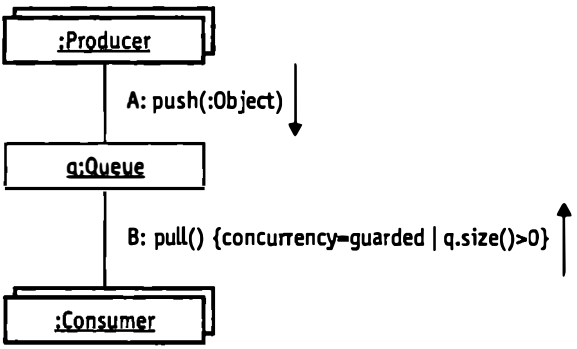


Рис. 9.27. Взаимодействие в шаблоне Producer-Consumer

РЕАЛИЗАЦИЯ

Некоторые версии реализации шаблона Producer-Consumer накладывают ограничения на максимальный размер очереди. В таких версиях реализации рассматривается специальный случай, когда очередь достигла своего максимального размера, а поток производителя намеревается поместить в очередь еще один объект. Тогда управление очередью обычно осуществляется при помощи шаблона Guarded Suspension, который вынуждает экземпляр класса `Producer`

ждать, пока экземпляр класса Consumer не заберет объект из очереди. Если для объекта, который экземпляр класса Producer хочет поместить в очередь, есть место, то ему разрешается закончить выполнение этой операции и продолжить свою дальнейшую работу.

СЛЕДСТВИЯ

☺ Объекты Producer могут поставлять производимые ими объекты объекту Queue, не дожидаясь объекта Consumer.

- Если в объекте Queue находятся объекты, объекты Consumer могут, не ожидая, достать объект из очереди. Однако если очередь пуста, а объект Consumer вызывает метод pull объекта Queue, то метод pull не выполняется до тех пор, пока объект Producer не поместит объект в очередь.

ПРИМЕНЕНИЕ В JAVA API

Ядро Java API содержит классы java.io.PipedInputStream и java.io.PipedOutputStream. Вместе они реализуют вариант шаблона Producer-Consumer, который называется шаблоном Pipe. Шаблон Pipe содержит только один объект Producer и только один объект Consumer. Этот шаблон обычно ссылается на объект Producer как источник данных и на объект Consumer как на приемник данных.

Классы java.io.PipedInputStream и java.io.PipedOutputStream совместно играют роль класса Queue. Они позволяют одному потоку записывать байтовый поток в другой поток. Потоки выполняют свои операции записи и чтения асинхронно по отношению друг к другу, если используемый ими внутренний буфер не пуст или не заполнен до предела.

ПРИМЕР КОДА

Следующие листинги содержат код, который реализует проект, рассмотренный в разделе «Контекст». Первые два листинга — это скелетные коды для классов Client и Dispatcher.

```
public class Client implements Runnable {
   private Queue myQueue;
   ...
   public Client(Queue myQueue) {
      this.myQueue = myQueue;
      ...
   } // constructor(Queue)
   ...
   public void run() {
```

```
      TroubleTicket tkt = null;
      ...
      myQueue.push(tkt);
   } // run()
} // class Client

public class Dispatcher implements Runnable {
   private Queue myQueue;
   ...
   public Dispatcher(Queue myQueue) {
      this.myQueue = myQueue;
   } // constructor(Queue)
   ...
   public void run() {
      TroubleTicket tkt = myQueue.pull();
      ...
   } // run()
} // class Dispatcher
```

Последний листинг содержит класс Queue.

```
public class Queue {
   private ArrayList data = new ArrayList();

   /**
    * Помещаем объект в конец очереди.
    * @param obj Объект, помещаемый в конец очереди.
    */
   synchronized public void push(TroubleTicket tkt) {
      data.add(tkt);
      notify();
   } // push(TroubleTicket)

   /**
    * Получаем TroubleTicket, находящийся в начале очереди.
    * Если очередь пуста, ждем, пока в ней не появится объект.
    */
   synchronized public TroubleTicket pull() {
      while (data.size() == 0){
         try {
            wait();
         } catch (InterruptedException e) {
```

```
            } // try
          } // while
          TroubleTicket tkt = (TroubleTicket)data.get(0);
          data.remove(0);
          return tkt;
        } // pull()

        /**
         * Возвращаем количество претензий, находящихся
         * в этой очереди.
         */
        public int size() {
          return data.size();
        } // size()
      } // class Queue
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ PRODUCER-CONSUMER

Guarded Suspension. Шаблон Producer-Consumer использует шаблон Guarded Suspension для управления такой ситуацией, когда объект `Consumer` ожидает получения объекта из пустой очереди.

Pipe. Шаблон Pipe — это специальный случай шаблона Producer-Consumer, который включает только один объект `Producer` и только один объект `Consumer`. Обычно шаблон Pipe ссылается на объект `Producer` как на источник данных и на объект `Consumer` как на приемник данных.

Scheduler. Шаблон Producer-Consumer может рассматриваться как специальная форма шаблона Scheduler, политика планирования которого имеет две характерные особенности:

- основана на доступности ресурса;
- назначает ресурс для потока, но не нуждается в повторном получении контроля над ресурсом, если поток завершен, поэтому он может переназначить ресурс другому потоку.

Two-Phase Termination (Двухфазное завершение)

СИНОПСИС

Обеспечивает нормальное завершение потока или процесса, устанавливая флаг. Поток или процесс проверяют значение флага в стратегических точках своего выполнения.

КОНТЕКСТ

Предположим, что нужно написать серверную часть приложения, предоставляющую логику среднего уровня для рабочей станции, используемой для торговли акциями. Клиент подключается к серверу и сообщает, что его интересуют некоторые акции. Сервер отправляет клиенту текущую цену на эти акции. Когда сервер получает информацию о том, что пакеты акций выставлены на продажу, он сообщает об этом заинтересованным клиентам.

Сервер создает поток для каждого клиента. Этот поток отвечает за предоставление информации о продаже акций клиенту, которого он обслуживает.

Сервер должен также реагировать на некоторые административные команды. Одна из таких команд выполняет отключение клиента. Когда на сервер поступает запрос отключить клиента, сервер должен закрыть поток, обслуживающий этого клиента, и высвободить соответствующие ресурсы, используемые потоком.

Кроме того, сервер должен реагировать на административную команду, которая отключает весь сервер.

Эти команды похожи с точки зрения выполняемых ими действий. Основное отличие заключается в масштабах действия: одна команда завершает только поток, другая — весь процесс. В обоих случаях способы реализации схожи. На рис. 9.28 представлена диаграмма взаимодействия, демонстрирующая такую организацию потока сервера, которая предполагает отключение только по запросу.

Взаимодействие начинается с вызова метода `run` объекта `Session`. Метод `run` сначала вызывает метод `initialize` объекта `Session`. Затем он периодически вызывает метод `sendTransactionsToClient` объекта `Portfolio`. Он продолжает вызывать этот метод, пока метод `isInterrupted` объекта `Session` возвращает false. Метод `isInterrupted` объекта `Session` будет возвращать false до тех пор, пока не будет вызван метод `interrupt` объекта `Session`.

Нормальная последовательность событий, завершающая сеанс, начинается не с того потока, который вызвал метод `run`, а с другого. Этот другой поток вызы-

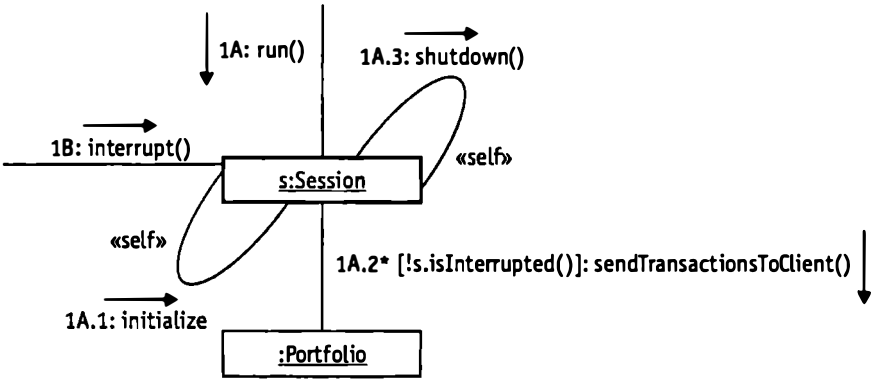


Рис. 9.28. Завершение потока сервера

вает метод `interrupt` объекта `Session`. Тогда при следующем вызове объектом `Session` своего метода `isInterrupted` этот метод возвратит true, и метод `run` прекратит вызывать метод `sendTransactionsToClient` объекта `Portfolio`. Затем он вызывает метод `shutDown` объекта `Session`, который выполняет все необходимые операции по закрытию.

Механизм нормального завершения процесса похож на механизм завершения потоков. Если получена команда завершить весь процесс, устанавливается флаг, который приводит к завершению каждого потока, участвующего в процессе.

МОТИВЫ

☺ Поток или процесс проектируются так, чтобы работать в течение неопределенного времени.

☺ Случай вынужденного завершения процесса или потока без возможности проведения необходимых действий по завершению работы является нежелательным.

☺ При поступлении к потоку или процессу запроса на завершение вполне допустимо, что перед завершением потока или процесса потребуется некоторое время для очистки.

РЕШЕНИЕ

Для закрытия потока или процесса применяются две основные технологии. Один способ состоит в том, чтобы завершить их сразу. Другой способ — запросить закрытие потока или процесса и затем ожидать его осуществления, предполагая выполнение необходимых действий по завершению работы и только потом — закрытия.

На рис. 9.29 показан класс, который можно использовать для согласования операций завершения процесса. Использование такого класса позволяет про-

цессу получать запрос на завершение и затем выполнять освобождение своих ресурсов перед непосредственным закрытием. Каждый поток процесса вызывает метод `isShutdownRequested` в стратегических точках своего выполнения. Если этот метод возвращает true, поток выполняет действия по освобождению ресурсов и завершается. При закрытии («смерти») всех потоков приложения происходит закрытие процесса.

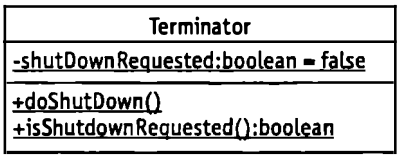


Рис. 9.29. Завершение процесса

Завершение отдельного потока предполагает использование предназначенного специально для него флага. Такой флаг имеет каждый поток в Java, так как флаг является частью класса `Thread`. Ему присваивается значение true при вызове метода `interrupt` потока. Значения флага можно получить, вызвав метод `isInterrupted` потока.

РЕАЛИЗАЦИЯ

После того как процесс или поток получил запрос на завершение, трудно, а порой и невозможно определить, будет ли процесс или поток реально завершен. Поэтому, если существует какая-то неопределенность в отношении завершения процесса или потока после получения ими запроса на это завершение, по истечении предварительно заданного времени процесс или поток должен быть принудительно закрыт.

Чтобы вынужденно завершить поток, можно вызвать его метод `stop`. Механизм вынужденного завершения процесса зависит от используемой операционной системы.

Методы, задающие для флага завершения значение true, не нуждаются в синхронизации. Установка флага является идемпотентной. Один или несколько потоков могут выполнять операцию (параллельно или не параллельно), и результат все же будет соответствовать установленному в true значению флага завершения.

СЛЕДСТВИЯ

☺ Использование шаблона Two-Phase Termination позволяет процессам и потокам выполнять самоочистку непосредственно перед завершением.

☹ Использование шаблона Two-Phase Termination может задержать выполнение процесса или потока на неопределенное время.

ПРИМЕНЕНИЕ В JAVA API

Ядро Java API не использует шаблон Two-Phase Termination, однако в нем имеются средства для поддержки этого шаблона.

Для поддержки двухфазного завершения потоков класс `Thread` предоставляет метод `interrupt`, запрашивающий завершение потока. Кроме того, класс `Thread` содержит метод `isInterrupted`, позволяющий потоку обнаруживать запрос на его завершение.

Существуют методы, например, `sleep`, которые приводят поток в состояние ожидания. Считается, что, если поток получил запрос на завершение, ожидая выполнения одного из таких методов, он завершится сразу после выполнения методов, приведших поток в состояние ожидания. Чтобы гарантировать своевременность закрытия потока, некоторые методы, заставляющие поток чего-то ожидать, генерируют исключение `InterruptedException`, если поток ждет выполнения одного из таких методов в момент вызова его метода `interrupt`. Генерировать исключение `InterruptedException` могут, например, следующие методы: `Thread.sleep`, `Thread.join` и `Object.wait`.

Что касается поддержки завершения процесса при отключении потока, то процесс будет завершен, если при этом не остается потоков, которые являются действующими и не являются потоками-демонами.

Обратите внимание, что Java не предоставляет какой-либо возможности непосредственного определения или приема сигналов или прерываний от операционной системы, которые могут привести к закрытию процесса. Класс `java.lang.Runtime` имеет метод `addShutdownHook`, который может быть использован для регистрации кода, который будет работать при завершении JVM. Документация, описывающая этот метод, обещает, что JVM приложит все усилия для работы кода. Но нет никакой гарантии, что он действительно будет работать.

ПРИМЕР КОДА

Следующий листинг содержит код, который реализует проект, рассмотренный в разделе «Контекст»:

```
public class Session implements Runnable {
   private Thread myThread;
   private Portfolio portfolio;
   private Socket mySocket;
...
   public Session(Socket s) {
      myThread = new Thread(this);
      mySocket = s;
```

```
      ...
   } // constructor()

   public void run() {
      initialize();
      while (!myThread.interrupted()) {
         portfolio.sendTransactionsToClient(mySocket);
      } // while
      shutDown();
   } // run()

   /**
    * Запрос на завершение этого сеанса.
    */
   public void interrupt() {
      myThread.interrupt();
   } // interrupt()

   /**
    * Инициализирует этот объект.
    */
   private void initialize() {
      ...
   } // initialized

   /**
    * Выполняем необходимые действия по освобождению ресурсов
    * для этого объекта.
    */
   private void shutDown() {
      //...
   } // shutDown()
...
} // class Session
```

Double Buffering (Двойная буферизация)

Этот шаблон известен также под названием Exchange Buffering.

СИНОПСИС

Шаблон Double Buffering подготавливает данные к применению потребителями, позволяя избежать задержек во время использования объектом, благодаря синхронному генерированию данных.

КОНТЕКСТ

Предположим, что нужно сопровождать ПО, которое загружает все транзакции системы торговых терминалов, выполненные за день, в базу данных, поддерживающую хранилище данных (warehouse). Задача программиста — уменьшить количество времени, необходимого для загрузки транзакций системы торговых терминалов в базу данных.

Оказывается, что серьезным препятствием повышению производительности является то, что программа тратит очень много времени на чтение файлов, содержащих записи транзакций. Для буферизации считываемых записей программа использует экземпляр класса `java.io.BufferedInputStream`. Чтобы уменьшить количество необходимых операций чтения, можно попытаться увеличить размеры буфера, используемого объектом `BufferedInputStream`.

В результате увеличения размеров буфера программа затрачивает меньше времени на чтение записей транзакций системы терминалов, но все еще ожидает считываемые из файла транзакции, перед тем как добавляет их в базу данных. Поскольку база данных и файлы транзакций хранятся на разных физических дисках, возможно параллельное считывание транзакций и обновление базы данных. Пока некоторые транзакции системы торговых терминалов добавляются в базу данных, желательно, чтобы следующие записи транзакций были бы уже считаны к тому времени, когда предыдущие транзакции системы терминалов будут добавлены в базу данных. Таким образом программе не нужно ждать, пока будут считаны следующие транзакции.

Чтобы решить эту проблему (когда программа должна ожидать чтения транзакций системы терминалов), создают класс, похожий на класс `BufferedInputStream`. Новый класс называется `DoubleBufferedInputStream` и будет использовать два буфера вместо одного. При первом вызове одного из методов чтения этого класса синхронно заполняется один из его буферов. С этого момента каждый буфер может исполнять одну из двух ролей.

Одна роль называется *активный буфер*; другая — *резервный буфер*. Буфер, который синхронно заполняется во время вызова первой операции чтения, сначала

является активным, а второй буфер — резервным. Как только активный буфер заполнится, происходит следующее:

- метод `read` считывает байты из активного буфера;
- в то же самое время байты асинхронно считываются из основного входного потока и поступают в резервный буфер.

Когда активный буфер пуст и асинхронное заполнение резервного буфера завершено, эти два буфера меняются ролями. Если чтение байтов в резервный буфер еще не завершилось, обмен ролями откладывается до того момента, пока не закончится чтение. После того как произошел обмен ролями, метод `read` считывает байты из активного буфера, а резервный буфер заполняется байтами, асинхронно считываемыми из основного входного потока. Эти взаимодействия показаны на рис. 9.30.

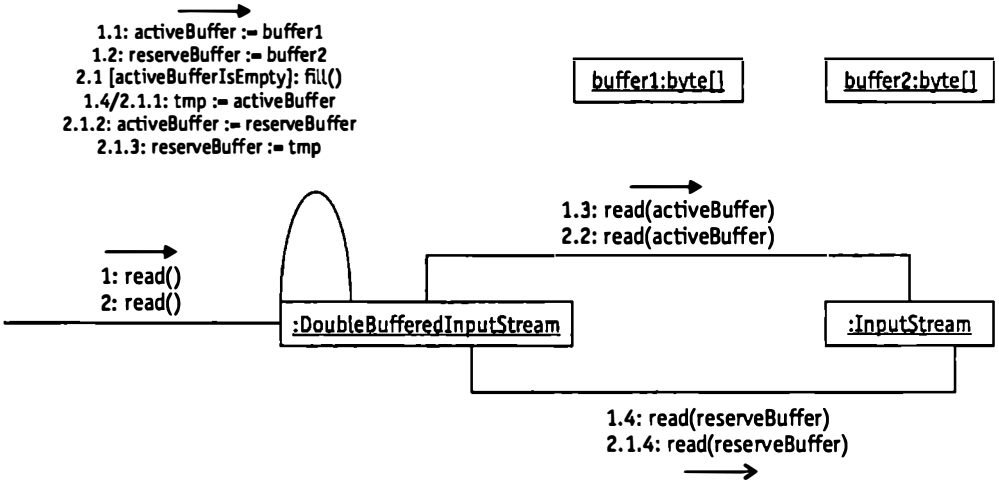


Рис. 9.30. Взаимодействие для класса DoubleBufferedInputStream

МОТИВЫ

☺ Объект использует много данных сразу.

☺ Данные получаются объектом-потребителем от объекта, создающего данные. Объект, производящий данные, не управляет тем, когда объект-потребитель потребует новых данных.

☺ Если объекту-потребителю нужны данные, а они недоступны, это в какой-то мере отражается на производительности объекта-потребителя.

☺ По прошествии некоторого времени средняя скорость использования данных не может превысить скорость их создания.

РЕШЕНИЕ

Поведение некоторых объектов, использующих данные, таково, что они используют все данные одного буфера, а потом им не требуются данные из другого буфера в течение неопределенного времени. Если это время в целом больше времени, необходимого для повторного заполнения буфера, то можно повторно использовать тот же самый буфер. Но если объект-потребитель не использует сразу все данные, находящиеся в буфере, или не ждет достаточно долго, что позволяет повторно заполнить буфер, то потребуется не один буфер.

На рис. 9.31 представлено взаимодействие между объектами, принимающими участие в шаблоне Double Buffering. Объект `DataProvider` предоставляет данные объекту, который вызывает его метод `getData`. Объекты `buffer1` и `buffer2` используются объектом `DataProvider` для временного хранения данных в памяти, начиная с момента их выборки или создания и заканчивая тем моментом, когда они возвращаются методом `getData`. Объект `DataSource` отвечает за выборку и создание данных.

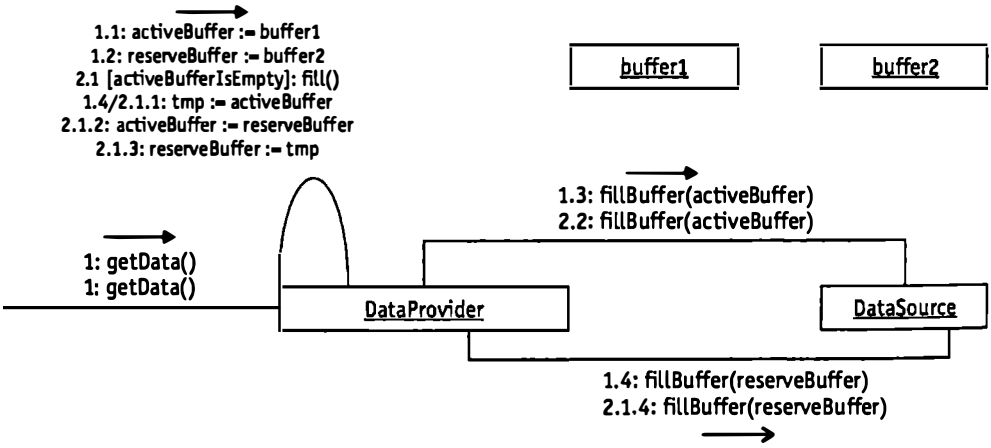


Рис. 9.31. Взаимодействие при двойной буферизации

Опишем эти взаимодействия.

1. Первый запрос на получение данных из произвольного источника.
   - **1.1.** Объект `buffer1` становится активным буфером.
   - **1.2.** Объект `buffer2` становится резервным буфером.
   - **1.3.** Активный буфер синхронно заполняется.
   - **1.4.** Резервный буфер асинхронно заполняется. Этот вызов метода заканчивает свое выполнение сразу, не ожидая заполнения резервного буфера. За заполнение резервного буфера отвечает отдельный поток.

2. Запрос на получение новых данных. Если в активном буфере находится достаточное количество данных, запрос удовлетворяется с помощью этих данных. В противном случае нужны дополнительные данные и выполняется пункт **2.1**. После завершения выполнения пункта **2.1** становятся доступными дополнительные данные, с помощью которых удовлетворяется запрос.

   2.1. Этот шаг выполняется только в том случае, если активный буфер пуст. При этом должно гарантироваться, что резервный буфер, который будет использоваться в качестве активного, заполнен, или заполнен настолько, сколько нужно для получения данных.

      2.1.1. Если асинхронное заполнение резервного буфера не завершено, то нужно подождать, пока оно закончится. Потом происходит обмен ролей буферов, активного и резервного.

      2.1.2, 2.1.3. Эти шаги являются завершающими при перемене ролей активного и резервного буферов. После шага **2.1.3** буфер, который был резервным перед шагом **2.1.1**, становится активным, а буфер, который был активным, становится резервным.

      2.1.4. Резервный буфер асинхронно заполняется. Вызов этого метода завершается сразу, не ожидая заполнения резервного буфера. За заполнение резервного буфера отвечает отдельный поток.

РЕАЛИЗАЦИЯ

При каждом запросе на получение данных используется некоторое количество данных. Буферы должны быть достаточно большими, чтобы содержащееся в них количество данных позволяло удовлетворить один запрос.

Многочисленные буферы

Одно из предположений, лежащих в основе шаблона Double Buffering, гласит, что с течением времени средняя скорость производства данных по меньшей мере становится равной скорости их использования. Некоторые приложения иногда могут много раз запрашивать данные в течение непродолжительного времени, а затем какое-то время вообще ничего не запрашивать. В подобных ситуациях данные иногда могут использоваться быстрее, чем может заполниться буфер. Чтобы избежать необходимости ожидания данных, когда на получение этих данных приходит группа запросов в пакетах, используют несколько буферов.

Дополнительные буферы применяются в качестве резервных. При достаточном количестве резервных буферов, заполненных заранее (до того как потребуется их содержимое), группы запросов на получение данных не должны будут ждать получения этих данных.

Управление исключительной ситуацией

Асинхронные операции выполняются с целью заполнения резервных буферов данными до того, как эти данные могут понадобиться. Такие операции могут генерировать исключения, которые требуют специальной обработки. С обработкой таких исключений связаны определенные трудности.

Проблема состоит в том, что эти исключения генерируются в потоке, который, возможно, был использован только для асинхронного опережающего чтения. Как правило, более неудобно и менее полезно управлять исключениями в потоке, выполняющем опережающее чтение, чем управлять исключениями в потоке, запрашивающем данные. Чтобы решить проблему, нужно перехватить исключения в потоке, выполняющем опережающее чтение, и заново сгенерировать их в потоке, запрашивающем данные. Однако при простой повторной генерации исключений обычно возникают проблемы с синхронизацией.

Когда в потоке, выполняющем опережающее чтение, генерируется исключение при получении данных, уже существуют данные, возвращенные в ответ на запросы. Если немедленно повторно сгенерировать исключение в потоке, запрашивающем данные, этот поток может не отбросить правильные данные, находящиеся между теми, которые он уже получил, и тем местом, где было сгенерировано исключение. Лучше не генерировать повторное исключение в потоке, запрашивающем данные, до тех пор пока этот поток не получит все данные вплоть до того места, в котором генерируется исключение.

Пример, рассмотренный в разделе «Пример кода», включает код, который обрабатывает исключения именно таким образом.

СЛЕДСТВИЯ

☺ Использование шаблона Double Buffering позволяет избегать таких ситуаций, когда приложение должно ждать получения новых данных, которые должны быть считаны или созданы перед тем, как продолжится их обработка.

• Использование шаблона Double Buffering приводит к увеличению объема памяти, используемого программой, поэтому она работает быстрее.

☹ Шаблон Double Buffering представляет собой оптимизацию, которая усложняет программу.

ПРИМЕНЕНИЕ В JAVA API

Java Swing использует шаблон Double Buffering. Большая часть компонентов Swing GUI представляет собой подклассы класса `javax.swing.JComponent`. Класс `JComponent` имеет метод `setDoubleBuffered`. Этот метод используется для включения и выключения двойной буферизации.

В разделе «Решение» описан специальный случай, когда при реализации шаблона Double Buffering нужен только один буфер. Особая ситуация, когда требу-

ется только один буфер, выглядит так: за один раз используется строго один буфер, заполненный данными, и промежуток времени между запросами на получение данных в целом больше, чем промежуток времени, необходимый для заполнения буфера. Библиотека Swing и пользовательские интерфейсы в общем случае могут служить примерами этого специального случая.

Swing реализует шаблон Double Buffering, используя для этой цели только один буфер, поскольку, даже если производятся быстрые обновления информации, изображаемой на экране, интервал между ними превышает несколько секунд. Это более чем достаточный промежуток времени для того, чтобы нарисовать экранное изображение в буфере перед тем, как оно может понадобиться.

ПРИМЕР КОДА

Примером кода для этого шаблона может служить подкласс класса `java.io.InputStream`. Он похож на класс `java.io.BufferedInputStream` за тем исключением, что вместо использования одинарной буферизации, позволяющей уменьшить количество операций физического чтения, необходимых для чтения файла, он применяет двойную буферизацию, тоже с целью отказа от необходимости ожидания операций физического чтения.

```
public
class DoubleBufferedInputStream extends FilterInputStream {
  private static final int DEFAULT_BUFFER_COUNT = 2;
  private static final int DEFAULT_BUFFER_SIZE = 4096;

  private byte[][] buffers;
```

Двойная буферизация реализуется при помощи массива массивов, на который ссылается переменная `buffers`. Активный буфер представлен тем массивом, индекс которого в массиве высшего уровня совпадает со значением переменной `activeBuffer`. Другие массивы используются в качестве резервных.

```
  private int activeBuffer = 0;

  /**
   * Количество байтов в каждом буфере.
   */
  private int[] counts;
```

При заполнении буферов этот класс пытается заполнить их полностью. Но это не всегда возможно. Реальное количество байтов данных, содержащихся в буфере, находится в элементе массива `counts`, соответствующем этому буферу.

```
  /**
   * Индекс следующего символа,
   * который должен быть считан из активного буфера.
```

```
     *
     * Справедливо следующее утверждение:
     *     0 <= pos <= counts[activeBuffer]
     *
     * Если pos==counts [activeBuffer],
     * то активный буфер пуст.
     */
    private int pos;

    /**
     * Если исключение генерируется во время опережающего чтения
     * с целью заполнения резервного буфера, этой переменной
     * присваивается объект исключения, чтобы он мог быть
     * повторно сгенерирован позже в потоке, запрашивающем данные,
     * когда этот поток достигнет точки, где было сгенерировано
     * исключение.
     */
    private Throwable exception;

    /**
     * Если эта переменная равна true,
     * достигнут конец входного потока.
    */
    private boolean exhausted;

    /**
     * Эта переменная становится равной true после того, как весь
     * основной входной поток был считан в буфер.
     */
    private boolean eof;

    /**
     * Этот объект отвечает за асинхронное
     * заполнение резервных буферов.
     */
    private BufferFiller inyBufferFiller;
```

Класс `BufferFiller` представляет собой закрытый класс, код которого приводится в конце данного листинга.

```
    /**
     * Объект блокировки, который используется для синхронизации
     * потоков, запрашивающих данные, и заполнения буферов.
     */
    private Object lockObject = new Object();
```

Следующий метод генерирует исключение IOException, если он вызывается после метода close объекта DoubleBufferedInputStream. Он вызывается одним из методов read или skip класса с тем, чтобы они сгенерировали исключение IOException в том случае, если эти методы были вызваны после закрытия потока. Этот метод проверяет, не равно ли null значение переменной buffers. В переменной buffers конструктором класса устанавливается ссылка на массив массивов; null в ней устанавливается методом close.

```
private void checkClosed() throws IOException {
   if (buffers == null) {
      throw new IOException("Stream closed");
   } // if
} // checkClosed()

/**
 * Создает объект DoubleBufferedInputStream,
 * который будет читать входные данные из этого
 * входного потока, используя заданный по умолчанию
 * размер буфера и два буфера.
 */
public DoubleBufferedInputStream(InputStream in) {
   this(in, DEFAULT_BUFFER_SIZE);
} // constructor(InputStream)

/**
 * Создает объект DoubleBufferedInputStream,
 * который будет читать
 * входные данные из этого входного потока,
 * используя два буфера заданного размера.
 */
public DoubleBufferedInputStream(InputStream in, int size){
   this (in, size, DEFAULT_BUFFER_COUNT);
} // constructor(InputStream, int)

/**
 * Создает объект DoubleBufferedInputStream,
 * который будет читать входные данные
 * из этого входного потока,
 * используя заданное количество буферов определенных
 * размеров.
 */
```

```
public DoubleBufferedInputStream(InputStream in,
                                 int size,
                                 int bufferCount){
   super(in);
   if (size < 1) {
      String msg = "Buffer size < 1";
      throw new IllegalArgumentException(msg);
   } // if size
   if (bufferCount < 2) {
       bufferCount = 2;
   } // if

   buffers = new byte[bufferCount][size];
   counts = new int[bufferCount];
   myBufferFiller = new BufferFiller();
} // constructor(InputStream, int, int)

/**
 * Возвращает следующий байт данных или -1,
 * если был достигнут конец потока.
 */
public synchronized int read() throws IOException {
   checkClosed();
   if (eof) {
      return -1;
   } // if eof
   if (pos >= counts[activeBuffer]) {
       eof = !advanceBuffer();
   } // if empty
   if (eof) {
      return -1;
   } // if eof

   // Возвращает результат операции (&)
   // над следующим символом и 0xff,
   // поэтому значения 0x80-0xff больше не рассматриваются как
   // отрицательные.
   int c = buffers[activeBuffer][pos++] & 0xff;
   if (pos >= counts[activeBuffer]) {
       eof = !advanceBuffer();
   } // if empty
   return c;
} // read()
```

```
/**
 * Читает заданное количество байтов из этого входного поток
 * в определенный байтовый массив, начиная с заданного
 * смещения.
 *
 * @return Возвращает количество считанных байтов или -1,
 *     если был достигнут конец потока.
 */
public synchronized int read(byte b[], int off, int len)
                                          throws IOException {
   checkClosed();
   if ((off | len | (off + len) | (b.length - (off + len))) < 0
      throw new IndexOutOfBoundsException();
   } else if (eof) {
      return -1;
   } else if (len == 0) {
      return 0;
   } // if
   int howMany = 0;
   do {
      if (len <= counts[activeBuffer]-pos) {
         System.arraycopy(buffers[activeBuffer], pos,
                          b, off, len);
         howMany += len;
         pos += len;
         len = 0;
      } else {
         int remaining = counts[activeBuffer]-pos;
         System.arraycopy(buffers[activeBuffer], pos,
                          b, off, remaining);
         howMany += remaining;
         pos += remaining;
         len -= remaining;
         off += remaining;
      } // if len
      if (pos >= counts[activeBuffer]) {
          eof = !advanceBuffer();
      } // if empty
   } while (!eof && len>0) ;
   return howMany;
} // read(byte[], int, int)
```

```
/**
 * Этот метод делает текущий активный буфер пустым
 * резервным буфером, а следующий резервный буфер -
 * новым активным буфером. Если следующий резервный буфер еще
 * не был полностью заполнен и есть основания полагать,
 * что он будет заполнен,
 * то этот метод ожидает заполнения.
 *
 * @return Возвращает true, если этот вызов был успешным
 *     при переходе к другому заполненному буферу.
 */
private boolean advanceBuffer() throws IOException {
   int nextActiveBuffer = (activeBuffer+1)%counts.length;
   if (counts[nextActiveBuffer]==0) {
      if (exhausted) {
         rethrowException();
         return false;
      } // if exhausted

      // Мы не знаем, существует ли текущий запрос
      // на заполнение резервного буфера или нет,
      // поэтому выясняем это.
      myBufferFiller.fillReserve();

      synchronized (lockObject) {
         while ( counts[nextActiveBuffer]==0
                 && !exhausted) {
            try {
                 lockObject.wait();
            } catch (InterruptedException e) {
            IOException ioe
               = new InterruptedIOException();
               ioe.initCause(e);
               throw ioe;
            } // try
         } // while
      } // synchronized

      if (counts[nextActiveBuffer]==0 && exhausted) {
         rethrowException();
         return false;
      } // if exhausted
   } // if ==0
```

```
    // Теперь нам известно, что следующий буфер содержит данные,
    // поэтому делаем его активным.
    counts[activeBuffer] = 0;
    activeBuffer = nextActiveBuffer;
    pos = 0;

    // Сейчас прежний активный буфер является пустым резервным
    // буфером, попытаемся его заполнить до того, как он нам
    // понадобится.
    myBufferFiller.fillReserve();
    return true;
} // advanceBuffer()

/**
 * В текущем потоке повторно сгенерируем исключение, которое
 * мы перехватили потоком, выполняющим опережающее чтение.
 */
private void rethrowException() throws IOException {
    if (exception!=null) {
        Throwable e = exception;
        exception = null;
        close();
        if (e instanceof IOException) {
            throw (IOException)e;
        } else if (e instanceof RuntimeException) {
            throw (RuntimeException) e;
        } else if (e instanceof Error) {
            throw (Error)e;
        } else {
            IOException ioe;
            ioe = new IOException("An error occurred");
            ioe.initCause(e);
            throw ioe;
        } // if
    } // if
} // rethrowException()

/**
 * Этот метод пропускает заданное количество байтов или,
 * возможно, меньше. Чтобы избежать проблемы, связанной с тем,
```

```
     * что этот метод должен ожидать завершения операции
     * опережающего чтения,
     * такая реализация метода не пропустит байтов больше,
     * чем их содержится в данный момент в активном и резервном
     * буферах.
     * @return Возвращает реальное количество пропущенных байтов.
     */
    public synchronized long skip(long n) throws IOException {
      checkClosed();
      if (n <= 0) {
        return 0;
      }
      long skipped = 0;
      int thisBuffer = activeBuffer;
      do {
        int remaining = counts[thisBuffer]-pos;
        if (remaining >= n) {
          pos += n;
            skipped += n;
            break;
          } // if
          pos = 0;
          n -= remaining;
          skipped += remaining;
          thisBuffer = (thisBuffer+1)%counts.length;
      } while (thisBuffer!=activeBuffer);

      activeBuffer = thisBuffer;
      myBufferFiller.fillReserve();
      return skipped;
    } // skip(long)

    /**
     * Возвращает количество байтов, которые могут быть прочитаны
     * из данного входного потока без блокировки.
     *
     * Эта реализация возвращает общее количество байтов данных
     * в активном и резервном буферах.
     */
    public synchronized int available() throws IOException {
```

```
    checkClosed();

    int count = 0;
    for (int i=0; i<counts.length; i++) {
       count += counts[i];
    } // for
    return count;
} // available()

/**
 * Закрывает этот входной поток и освобождает
 * все системные ресурсы, связанные с этим потоком.
 */
public void close() throws IOException {
    if (buffers == null)
       return;
    myBufferFiller.close();
    buffers = null;
    counts = null;
} // close

/**
 * Этот внутренний класс отвечает
 * за асинхронное заполнение резервных буферов.
 */
private class BufferFiller implements Runnable {
 /**
  * Эта переменная равна true после завершения обращения
  * к fillReserve и остается true до тех пор,
  * пока не закончится запрошенное асинхронное
  * заполнение резервных буферов.
  */
 private boolean outstandingFillRequest;

 /**
  * Этот поток отвечает
  * за асинхронное заполнение резервных буферов.
  */
  private Thread myThread;

  BufferFiller() {
     myThread = new Thread(this, "Buffer Filler");
```

```
        myThread.start();
    } // constructor()

/**
 * Синхронное заполнение одного буфера.
 * Полагаем, что все вопросы синхронизации были
 * решены тем, кто вызывает этот метод.
 *
 * @param ndx
 *     Индекс заполняемого буфера.
 * @return Возвращает количество байтов, считанных в буфер.
 */
    private int fillOneBuffer(int ndx) throws IOException {
        counts[ndx] = in.read(buffers[ndx]);
        return counts[ndx];
    } // fillOneBuffer(int)

/**
 * Заполняем любые пустые резервные буферы.
 */
    private void fill() throws IOException {
        for ( int i = (activeBuffer+1)%counts.length;
              i != activeBuffer
                   && !myThread.isInterrupted();
              i = (i+1)%counts.length) {
            if (counts[i]==0) {
                int thisCount = fillOneBuffer(i);
                if (thisCount == -1) {
                // конец файла
                exhausted = true;
                Thread.currentThread().interrupt();
            } else {
                // Оповещает любой поток,
                // ожидающий самого последнего заполненного буфера.
            synchronized (lockObject) {
                    lockObject.notifyAll() ;
                } // synchronized
                } // if eof
            } // if ==0
        } // for
    } // fill()
```

```
/**
 * Это общая логика для предварительного
 * заполнения резервных буферов.
 */
  public synchronized void run() {
     try {
       while ( !myThread.isInterrupted()
               && !exhausted){
          synchronized (this) {
               while (!outstandingFillRequest) {
                     wait();
               } // while
          } // synchronized
          fill();
          outstandingFillRequest = false;
       } // while
     } catch (InterruptedException e) {
       // Ничего не делает. Это нормально.
     } catch (ThreadDeath e) {
       throw e;
     } catch (Throwable e) {
       exception = e;
     } finally {
       exhausted = true;

          // Оповещает любой поток,
          // ожидающий окончания заполнения.
       synchronized (lockObject) {
          lockObject.notifyAll();
       } // synchronized
       try {
          in.close();
       } catch (IOException e) {
          if (exception==null) {
              exception = e;
          } // if
       } // try
       in = null;
     } // try
  } // run()
```

```
    /**
     * Запрашивает асинхронное заполнение всех резервных буферов.
     * Если операция асинхронного заполнения уже приведена
     * в действие, то вызов этого метода не принесет
     * никакого эффекта.
     */
    synchronized void fillReserve() {
        outstandingFillRequest = true;
        notify();
    } // fillReserve

    /**
     * Завершает асинхронное заполнение буфера.
     */
    void close() {
        myThread.interrupt();
    } // close()
    } // class BufferFiller
} // class DoubleBufferedInputStream
```

Что касается класса `DoubleBufferedInputStream`, то, поскольку его экземпляры имеют связанные с ними потоки, они никогда не будут удалены при сборке мусора до тех пор, пока не будут закрыты. Поток закрывается методом `close`.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ DOUBLE BUFFERING

Producer-Consumer. Шаблон Double Buffering представляет собой специальную форму шаблона Producer-Consumer.

Guarded Suspension. Шаблон Guarded Suspension используется при реализации шаблона Double Buffering для координации действий потоков, запрашивающих данные, с потоком, выполняющим опережающее чтение.

Two-Phase Termination. Шаблон Two-Phase Termination может быть использован при реализации шаблона Double Buffering для надлежащего завершения потока, выполняющего опережающее чтение.

Asynchronous Processing (Асинхронная обработка)

СИНОПСИС

Объект получает запросы на выполнение каких-либо действий. Шаблон Asynchronous Processing не позволяет асинхронно обрабатывать процессы. Вместо этого они ставятся в очередь и затем обрабатываются асинхронно.

КОНТЕКСТ

Поскольку запросы на выполнение действий над объектами бывают разными, описание этого шаблона включает два разных сценария, которые позволяют охватить всю широту проблемы, решаемой с помощью этого шаблона: сценарий для сервера и сценарий для клиента. Сначала рассмотрим сценарий, предназначенный для сервера.

Предположим, что проектируется сервер, который будет создавать стандартные письма от имени приложений. Ожидается, что приложения будут передавать объект серверу, который содержит информацию, необходимую для создания стандартного письма. Если сервер определяет, что объект содержит правильные данные, он возвращает идентификационный номер в приложение. Позже идентификационный номер может быть использован приложением для получения созданного письма и его отправки.

Самый простой способ работы такой программы предполагает синхронное создание стандартного письма. Это означало бы, что сервер имеет поток, получающий запрос, создает стандартное письмо, возвращает ID этого письма в то приложение, которое выдало запрос, а затем ожидает следующего запроса. Такие взаимодействия представлены на рис. 9.32. Однако в этом проекте существуют некоторые проблемы, которые вынуждают искать альтернативный вариант решения.

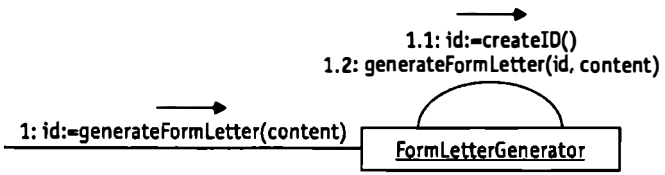


Рис. 9.32. Синхронная версия создания письма

Первая проблема связана с производительностью. С точки зрения клиентов, самый простой способ работы с сервером писем заключается в том, чтобы

передавать серверу данные для стандартного письма, а затем ждать, пока сервер не возвратит ID этого письма. Чем дольше приложение вынуждено ждать возврата сервером ID письма, тем больше полезных действий не могут быть сделаны во время этого ожидания. Если сервер обрабатывает запросы синхронно, то никто не может предугадать, сколько времени клиент может потратить на ожидание сервера.

Приложения могут справляться с этой проблемой следующим образом: активизировать поток, единственной целью которого является ожидание ID стандартного письма. Подобное решение вопроса позволяет приложениям избежать необходимости ожидания сервера, но за счет дополнительного усложнения каждого приложения, связанного с потребностью координирования дополнительного потока.

Другая проблема синхронизированного проекта выглядит более серьезной. Она обнаруживается в тот момент, когда сервер получает пакет запросов, прибывающих примерно в одно и то же время. Чтобы обслужить каждый запрос синхронно, сервер должен иметь, по меньшей мере, столько активных потоков, сколько существует текущих запросов.

Если сервер не ограничивает количество потоков, которым позволено работать параллельно, то достаточно большой пакет запросов приведет к перегрузке сервера. Если количество потоков превысит лимит, производительность быстро упадет. Кроме того, достаточно большое количество потоков (возможно, несколько сотен) будет причиной нехватки памяти для сервера.

Если сервер ограничивает количество потоков, работающих параллельно, то проблема может возникнуть в том случае, если сервер уже использует максимальное количество потоков. Поскольку сервер нуждается в отдельном потоке для каждого обрабатываемого им запроса, ему приходится отказываться обрабатывать запросы до тех пор, пока не завершатся некоторые текущие запросы и потоки, обрабатывающие их, станут доступными для обработки других запросов.

Альтернативой синхронной обработке запросов может служить асинхронная обработка (рис. 9.33).

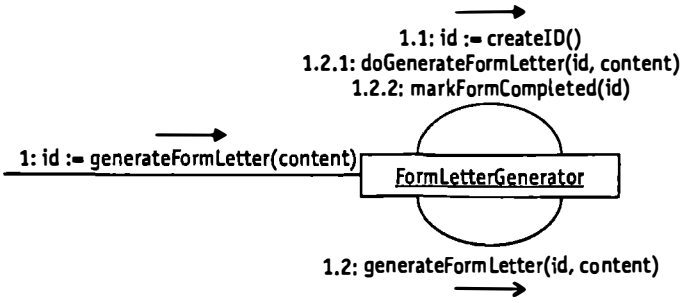


Рис. 9.33. Асинхронная версия создания письма

Опишем эти взаимодействия.

1. Клиент просит сервер создать стандартное письмо на основании тех данных, которые содержатся в заданном объекте. Этот вызов возвращается после того, как стандартному письму был присвоен идентификационный номер, но, как правило, до того как создано стандартное письмо.

   1.1. Сервер присваивает ID стандартному письму до его создания.

   1.2. Сервер инициирует процесс асинхронного создания стандартного письма.

      1.2.1. Создается стандартное письмо.

      1.2.2. Стандартное письмо помещается в базу данных, где над ним могут быть выполнены дальнейшие действия.

Делая создание стандартного письма асинхронным, решают проблемы, связанные с синхронной обработкой. Клиенты должны теперь ждать всего лишь столько времени, сколько требуется для присвоения ID стандартному письму. Сервер может оставить только один поток, который будет участвовать в создании стандартного письма. Если этот поток не может сразу обрабатывать запрос, он может поставить запрос в очередь, используя для этой цели шаблон Producer-Consumer. Применение асинхронной обработки немного усложняет сервер, но сервер — это единственное место, которое становится более сложным. Все его клиенты избавлены от подобной сложности.

Теперь рассмотрим сценарий для клиента.

Предположим, проектируется GUI с использованием Swing. Ожидается, что GUI должен выглядеть так, как показано на рис. 9.34.

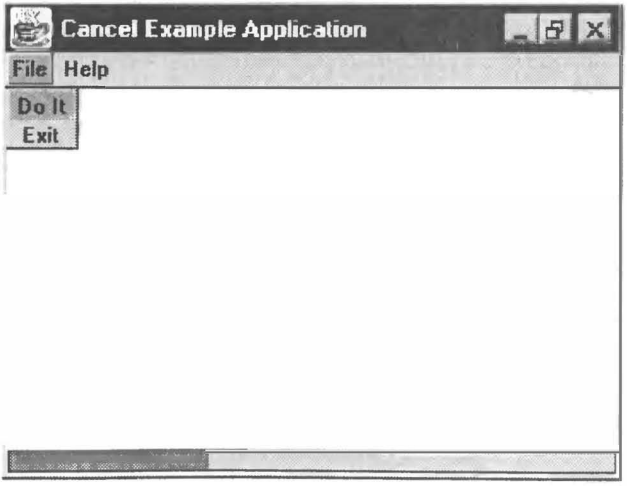


Рис. 9.34. Простой GUI

Если щелкнуть на пункте меню Do It, начинает выполняться длительная операция продолжительностью несколько секунд. Если операция не будет работать асинхронно по отношению к событию GUI, которое ее запустило, и GUI, и операция не будут себя вести надлежащим образом. Проблема возникает из-за того, что передача событий Swing является последовательной. Если в ответ на событие GUI команда работает синхронно, то ни одно, ни другое событие GUI не сможет быть обработано до тех пор, пока эта команда не завершится.

Первое, что можно отметить при использовании Swing GUI в том случае, если длительная команда работает синхронно, — это отсутствие обновления экрана до тех пор, пока не завершится команда. Это объясняется тем, что в Swing обычно обновляется экран в ответ на событие обновления экрана. Кроме того, события обновления экрана могут генерироваться при обращении к методу `refresh` компонента Swing.

После того как пользователь щелкает на пункт выпадающего меню, это меню обычно исчезает. Если длительная команда синхронно выполняется в ответ на выбор пункта выпадающего меню, это меню не исчезнет, пока команда не закончит выполнение. Это объясняется тем, что следующее событие обновления экрана не произойдет до тех пор, пока не завершится команда.

Swing позволяет команде работать асинхронно (и без использования другого потока) по отношению к тому событию, которое активизировало эту команду. Класс `javax.swing.SwingUtilities` имеет метод `invokeLater`, предназначенный специально для этой цели. Данный метод генерирует специальное событие, которое непосредственно запускает некоторый фрагмент кода. Это событие ставится в очередь подобно любому другому событию, и оно выполняется, когда достигает начала очереди. Вызов команды таким способом при выборе пункта меню позволяет выполняться событию обновления экрана (которое уже находится в очереди событий) до того, как будет выполнена команда. При этом решается проблема неисчезновения выпадающего меню.

Если команда делает что-то такое, что, как предполагается, должно изменять внешний вид пользовательского интерфейса, то ее запуск при помощи метода `invokeLater` — не самый лучший выход. Она должна работать в отдельном потоке. Предположим, например, что команда показывает на экране и периодически обновляет индикатор выполнения. Если эта команда вызывается при помощи метода `invokeLater`, то индикатор выполнения не появится до тех пор, пока не завершится эта команда. Когда индикатор выполнения наконец появится, он будет отражать только последнее состояние, в которое его установила команда. Это объясняется тем, что экран не обновлялся до тех пор, пока не закончилась команда. Чтобы сделать индикатор выполнения видимым и визуально обновляемым в процессе работы команды, эта команда должна выполняться в отдельном потоке (а не в том, который передает события).

Некоторые читатели могут заметить, что модальные диалоговые окна являются исключениями из этого правила (которое гласит, что команда должна выполняться в отдельном потоке, чтобы сделать видимыми изменения внешнего вида GUI в процессе выполнения команды). Команда может синхронно вызывать

модальное диалоговое окно. Когда диалоговое окно прекращает свою работу команда, инициировавшая диалог, продолжает выполнение и завершается.

Но это исключение кажущееся. Swing реализует модальные диалоги таким об разом, что во время работы модальных диалогов для обработки событий поль зовательского интерфейса запускается новый поток.

МОТИВЫ

☺ Предполагается, что объект должен отвечать на запросы с целью их обра ботки.

☺ Клиенты объекта, возможно, не должны ждать, пока он ответит на запрос.

☺ Запросы поступают асинхронно по отношению друг к другу.

☺ Объект может получать запрос в то время, когда он все еще занят ответом н предыдущий запрос.

☺ Может не быть фактического ограничения на количество запросов, посту пающих примерно в одно и то же время.

☹ Объект должен отвечать на запрос в течение определенного промежутк времени, начиная с момента, когда он получил запрос.

РЕШЕНИЕ

Проектируем объекты таким образом, чтобы обработка запросов происходил асинхронно по отношению к их получению. Общая схема такого решени представлена на рис. 9.35.

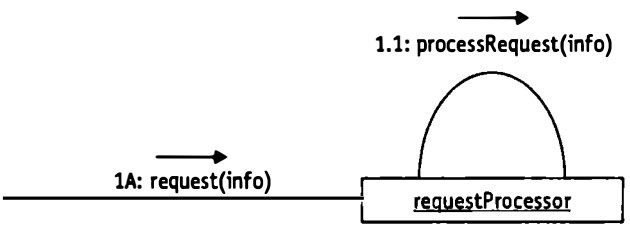


Рис. 9.35. Асинхронная обработка

С концептуальной точки зрения этот шаблон является самым простым сред всех шаблонов, описанных в этой книге. Однако, подобно многим другим кон цептуально простым идеям, он включает многочисленные реализационные де тали, которые должны быть тщательно разработаны.

РЕАЛИЗАЦИЯ

При проектировании и реализации шаблона Asynchronous Processing следует учитывать два основных вопроса. Первый вопрос, как будут управляться запросы и как будут распределены обрабатывающие их потоки. Второй вопрос возникает тогда, когда другие объекты должны знать о том, что запрос был обработан, или о том, каков результат обработки запроса.

Управление запросами и распределение потоков

Существует множество способов, позволяюших управлять запросами и распределять потоки, которые их обрабатывают. Под управлением запросов здесь понимается определение того, когда запрос не может быть обработан. Возможны следуюшие варианты:

- запрос немедленно может быть направлен на обработку;
- обработка запроса откладывается на некоторое определенное время;
- обработка откладывается до тех пор, пока не будет выполнено некоторое условие;
- запрос отклоняется и поэтому никогда не будет обработан.

Под распределением потока для обработки запроса здесь понимается определение того, когда запрос может быть обработан, что предусматривает выделение потока для обработки процесса и может также включать задание приоритета этого потока.

Самый простой способ согласования запросов с потоками состоит в том, чтобы запускать новый поток каждый раз, когда должен быть обработан запрос. Основное преимущество такого подхода — его простота. Недостатком является то, что этот способ обеспечивает минимальный контроль над управлением запросами и выделением потоков. Такой подход позволяет отбрасывать запрос, но не способен отложить его обработку. Он не обеспечивает никакой реальной политики распределения потоков или ограничения количества потоков. Возникнет проблема, если существует такая возможность, когда многочисленные извещения о событиях поступят примерно в одно и то же время. Если слишком большое количество событий обрабатывается одновременно, то общее количество потоков может привести к замедлению обработки запросов или прервать ее ввиду недостатка ресурсов.

Простая политика распределения потоков заключается в том, чтобы иметь постоянное количество потоков, пригодных для обработки событий, и произвольным образом назначать события потокам, по мере того как один из них становится доступным. Это, по сути, является применением шаблона Producer-Consumer. Поток, ответственный за получение запросов и помещение их в очередь, является производителем, а потоки, обрабатывающие запросы, являются потребителями.

Более сложная политика распределения потоков может быть реализована с помощью шаблона Thread Pool (описанного в книге [Grand2001]). Передача за-

просов пулу потоков позволяет реализовать более сложную политику распределения потоков. В ответ на некоторое требование можно изменять количество используемых потоков. Кроме того, в зависимости от сути запроса может изменяться приоритет потоков.

Управление запросами может быть реализовано с помощью шаблона Scheduler. Смысл объекта-планировщика в том, что он запрещает выполнение потока до тех пор, пока не будет выполнено некоторое условие. Применение объекта-планировщика позволяет отложить обработку запроса на основании практически любого критерия.

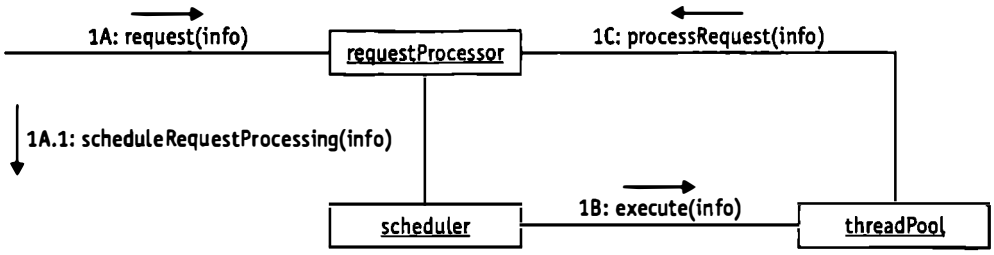


Рис. 9.36. Обработка запроса с использованием планировщика и пула потоков

На рис. 9.36 показаны взаимодействия между объектом-планировщиком и пулом потоков. При использовании простой очереди вместо пула потоков взаимодействия будут почти такими же.

1A. Запрос передается объекту `requestProcessor`.

1A.1. Объект `requestProcessor` передает информацию о запросе объекту `scheduler`.

1B. Когда объект `scheduler` решит, что настало время обработать запрос, он передает информацию о запросе объекту `threadPool`.

1C. Объект `threadPool` выделяет поток для обработки запроса, а объект `requestProcessor` предоставляет необходимую логику и информацию о состоянии.

Объект `scheduler` является активным, имеет свой собственный поток. Он способен распознавать, когда запрос готов к обработке, и передает его пулу потоков независимо от тех действий, которые выполняют любые другие потоки.

Если нужно сохранить потоки, можно реализовать объект `scheduler` как пассивный объект, который не имеет собственного потока. Тогда каждый раз при передаче запроса объекту `scheduler` этот объект будет проверять, имеются ли у него какие-либо запросы, готовые к передаче в пул потоков. Когда один из потоков пула завершает обработку некоторого запроса, он может (но не обязан) вызвать метод объекта `scheduler`, который проверит наличие запросов, готовых к обработке.

Управление результатом

В некоторых случаях после окончания обработки запроса другой объект может обнаружить, что обработка запроса завершена. Чаще всего объект, который, как полагают, должен быть осведомлен об окончании обработки запроса, — это тот объект, который выдал запрос. Существуют два самых распространенных способа, позволяющих другим объектам узнать о том, что обработка некоторого запроса завершена.

1. Объект, отвечающий за окончание обработки запроса, может отправлять событие тем объектам, которые были зарегистрированы как получатели таких событий.
2. Результат обработки события может быть сохранен в таком месте, в которое заинтересованные объекты могут отправлять запросы с целью обнаружения запросов и их результатов.

Если считается, что объект, который выдал запрос, узнал о результате обработки запроса, он может использовать шаблон Future для согласования результата обработки запроса с любыми другими действиями, в которых он принимает участие.

СЛЕДСТВИЯ

☺ Объект, являющийся источником запроса, не должен ждать окончания обработки этого запроса, поэтому он может выполнять другие действия, например, отправлять следующие события.

☺ Запрос может быть поставлен в очередь при отсутствии выделенного для него потока. Это большая экономия ресурсов, поскольку каждый поток использует значительный объем памяти, даже если он ничего не делает.

☺ Шаблон Asynchronous Processing позволяет проводить явную политику определения того, когда запрос будет обработан или будет ли он обработан вообще.

☹ Шаблон Asynchronous Processing не позволяет с уверенностью гарантировать, что запрос будет обработан в течение некоторого определенного времени.

ПРИМЕНЕНИЕ В JAVA API

Пользовательские интерфейсы, основанные на Swing и AWT, используют шаблон Asynchronous Processing. Они управляют событиями клавиатуры и мыши, применяя шаблон Producer-Consumer, и осуществляют их последовательное управление, используя единственный поток.

Процесс работает следующим образом. Когда пользователь делает что-нибудь с клавиатурой или мышью, платформно-зависимый механизм создает соответствующий объект, который описывает это событие. Затем он помещает событие

в очередь. Объект очереди является экземпляром класса `java.awt.EventQueue`. Объект `EventQueue` имеет связанный с ним поток, который достает объекты событий из очереди и распределяет их так, чтобы они передавались соответствующим получателям.

Объект `EventQueue`, используемый для передачи сообщений клавиатуры или мыши любому из компонент Swing или AWT, можно получить при помощи следующего кода:

```
import java.awt.Component;
java.awt.EventQueue
...
   Component c;
   EventQueue evtQueue;
...
   evtQueue = c.getToolkit().getSystemEventQueue();
```

ПРИМЕР КОДА

Пример кода для этого шаблона представляет собой клиентскую программу, асинхронно обрабатывающую некоторые события пользовательского интерфейса. Этот подкласс класса `JFrame` из библиотеки Swing создает элемент интерфейса, показанный на рис. 9.34. В его выпадающем меню File содержится пункт Do It. Кроме того, в нижней части фрейма находится индикатор выполнения.

Когда пользователь щелкает на пункт меню Do It, индикатор отображает ход выполнения процесса на протяжении примерно четырех секунд. В то время, пока индикатор выполнения изменяется, пункт Do It в меню File меняется на пункт Stop Doint It. Если пользователь щелкнет на пункт меню Stop Doing It, то индикатор выполнения перестанет изменяться.

Независимо от того, остановился ли индикатор выполнения из-за того, что он достиг 100 %, или из-за того, что пользователь щелкнул на пункт меню Stop Doing It, но, когда индикатор перестает показывать изменения, пункт меню Stop Doing It заменяется пунктом меню Do It.

```
public class CancelFrame extends JFrame {
   JMenuItem menuHelpAbout = new JMenuItem();
   JMenu menuHelp = new JMenu();
   JMenuItem menuFileDoIt = new JMenuItem();
   JMenuItem menuFileStopDoingIt = new JMenuItem();
   JMenuItem menuFileExit = new JMenuItem();
   JMenu menuFile = new JMenu();
   JMenuBar menuBar1 = new JMenuBar();
   BorderLayout borderLayout1 = new BorderLayout();
```

```
JProgressBar myProgress = new JProgressBar();
Thread doItThread;

public CancelFrame() {
   try {
        jbInit();
   } catch(Exception e) {
        e.printStackTrace();
   } // try
} // constructor()

/**
 * Инициализирует содержимое этого фрейма.
 */
private void jbInit() throws Exception {
   this.setJMenuBar(menuBar1);
   this.getContentPane().setLayout(borderLayout1);
   this.setSize(new Dimension(400, 300));
   this.setTitle("Cancel Example Application");
   menuFile.setText("File");
   menuFileDoIt.setText("Do It");
   menuFileDoIt.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
           fileDoIt_ActionPerformed(ae);
        } // actionPerformed(ActionEvent)
     });
   menuFileStopDoingIt.setText("Stop Doing It");
   menuFileStopDoingIt.addActionListener(
           new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
           fileStopDoingIt_ActionPerformed(ae);
        } // actionPerformed(ActionEvent)
     });
   menuFileExit.setText("Exit");
   menuFileExit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
           fileExit_ActionPerformed(ae);
        } // actionPerformed(ActionEvent)
     });
   menuHelp.setText("Help");
   menuHelpAbout.setText("About");
```

```
    menuHelpAbout.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent ae) {
             helpAbout_ActionPerformed(ae);
          } // actionPerformed(ActionEvent)
       });
    menuFile.add(menuFileDoIt);
    menuFile.add(menuFileExit);
    menuBar1.add(menuFile);
    menuHelp.add(menuHelpAbout);
    menuBar1.add(menuHelp);
    this.getContentPane().add(myProgress,
          BorderLayout.SOUTH);
} // jbInit()

/**
 * Асинхронно устанавливает индикатор выполнения в ноль,
 * а затем постепенно в течение нескольких секунд изменяет
 * его показание до 100 процентов.
 */
void fileDoIt_ActionPerformed(ActionEvent e) {
    doItThread = new Thread() {
       public void run() {
          try {
             installStopItMenuItem();
             myProgress.setMinimum(0);
             myProgress.setMaximum(19);
             myProgress.setValue(0);
             myProgress.repaint();
             for (int i=0; i<20; i++) {
                Thread.currentThread().sleep(300);
                myProgress.setValue(i);
                myProgress.repaint();
             } // for
          } catch (InterruptedException e) {
          } finally {
             installDoItMenuItem();
             doItThread = null;
          } // try
       } // run()
    };
    doItThread.start();
} // fileDoIt_ActionPerformed(ActionEvent)
```

```
    private void installStopItMenuItem() {
       menuFile.remove(menuFileDoIt);
       menuFile.insert(menuFileStopDoingIt, 0);
       menuFile.repaint();
    } // installStopItMenuItem()

    private void installDoItMenuItem() {
       menuFile.remove(menuFileStopDoingIt);
       menuFile.insert(menuFileDoIt, 0);
       menuFile.repaint();
    } // installDoItMenuItem()

    void fileStopDoingIt_ActionPerformed(ActionEvent e) {
       if (doItThread != null) {
          doItThread.interrupt();
       } // if
    } // fileStopDoingIt_ActionPerformed(ActionEvent)

    void fileExit_ActionPerformed(ActionEvent e) {
       System.exit(0);
    } // fileExit_ActionPerformed(ActionEvent)

    void helpAbout_ActionPerformed(ActionEvent e) {
       JPanel aboutPanel = new CancelFrame_AboutBoxPanel1();
       JOptionPane.showMessageDialog(this,
                aboutPanel,
                "About",
                JOptionPane.PLAIN_MESSAGE);
    } // helpAbout_ActionPerformed;
} // class CancelFrame
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ : ШАБЛОНОМ ASYNCHRONOUS PROCESSING

hread Pool. Шаблон Thread Pool (описанный в книге [Grand2001]) может использоваться в реализации шаблона Asynchronous Processing.

roducer-Consumer. Шаблон Producer-Consumer может использоваться в реализации шаблона Asynchronous Processing.

cheduler. Шаблон Scheduler может использоваться в реализации шаблона synchronous Processing.

acade. Реализация класса обработки запроса может применять классы, которые вляются продуктами использования шаблонов Thread Pool, Producer-Consumer

или Scheduler. Классы обработки запросов обычно выступают в роли фасада, скрывающего эти детали от клиентских классов.

Future. Объект, который выдает запрос, может нуждаться в том, чтобы знать о результате запроса, асинхронно обработанном другим объектом. Объект, выдающий запрос, может использовать шаблон Future для согласования результата обработки с любыми другими действиями, в которых участвует этот объект.

Active Object. Шаблон Active Object, рассмотренный в работе [SSRB00], описывает способ объединения шаблона Future с шаблоном Asynchronous Processing.

Future (Будущее)

Этот шаблон известен также под названием Promise.

СИНОПСИС

Используется объект, который инкапсулирует результат вычислений и позволяет скрыть от клиентов, синхронно или асинхронно выполняется вычисление. Этот объект будет содержать метод чтения результатов вычислений. Метод ожидает получения результата, если вычисления производятся асинхронно, и продолжает или выполняет вычисление, если оно является синхронным и еще не завершено.

КОНТЕКСТ

Предположим, что проектируется класс, который позволит программе получать текущие данные о погоде для данной местности. Ожидается, что общей сферой применения такого класса является отображение информации о погоде среди некоторой другой информации. Например, такой класс может использоваться как часть сервлета, который генерирует код HTML для Web-страницы, показывающей информацию о погоде наряду с новостями.

Время, которое понадобится для получения информации о погоде, будет меняться в значительной степени. Оно может составлять долю секунды или, возможно, несколько минут. Нужно спроектировать этот класс таким образом, чтобы минимизировать влияние на клиентов этого класса.

Можно создать экземпляры класса погоды, которые будут отправлять события заинтересованным объектам в том случае, когда этот класс получает затребованную информацию о погоде. Однако такой подход нежелателен, потому что если клиенты класса погоды должны будут получать асинхронный вызов, извещающий о наступлении события, то клиенты класса сильно усложнятся. Класс клиента вынужден будет реализовать метод по получению новой информации о погоде таким образом, чтобы обеспечить безопасность потока выполнения.

Лучше использовать структуру, показанную на рис. 9.37. Опишем классы, представленные на рис. 9.37.

Weather. Экземпляры этого класса инкапсулируют информацию о текущих погодных условиях определенной местности.

Client. Экземпляры класса, выступающего в этой роли, используют информацию, находящуюся в объектах `Weather`.

WeatherRequester. Когда объект `Client` хочет получить объект `Weather`, содержащий текущую информацию для данной местности, он просит объект

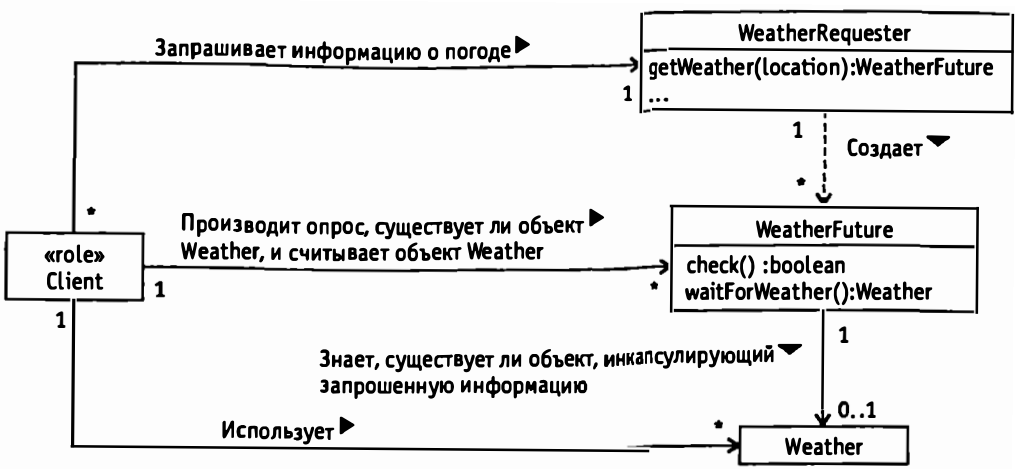


Рис. 9.37. Классы, предоставляющие информацию о погоде

`WeatherRequester` запросить эту информацию от его имени. Объект `Client` делает это, вызывая метод `getWeather` объекта `WeatherRequester`. Метод `getWeather` не ждет, пока считается информация о погоде. Он заканчивает выполнение немедленно. Возвращаемое им значение — это объект `WeatherFuture`. Запрошенная информация о погоде считывается асинхронно.

WeatherFuture. Этот класс отвечает за то, чтобы позволить другим потокам согласовывать свои действия с потоком, который считывает информацию о погоде. При вызове метод `getWeather` объекта `WeatherRequester` инициирует асинхронное считывание запрошенной информации о погоде и немедленно возвращает объект `WeatherFuture`, связанный с этим запросом.

Объект `WeatherFuture` имеет метод `check`, который возвращает false, если он вызывается до того, как была считана запрошенная информация и был создан объект `Weather`, содержащий эту информацию. После создания объекта Weather метод `check` возвращает true.

Объект `WeatherFuture` содержит еще один метод — `waitForWeather`. Если этот метод вызывается до того, как был создан объект `Weather`, он ожидает создания этого объекта. Как только объект `Weather` будет создан, метод `waitForWeather` немедленно возвращает этот объект `Weather`.

МОТИВЫ

☺ Вычисление производится асинхронно по отношению к другим потокам, использующим результат этого вычисления.

☺ Асинхронное вычисление может оповещать объект, использующий результат этого вычисления, о том, что результат уже доступен, при помощи шаблона Observer или генерируя для заинтересованного объекта некоторое событие.

Чтобы такой механизм работал надлежащим образом, может возникнуть необходимость в том, чтобы иметь однопоточный доступ к некоторым методам или сегментам кода. В некоторых случаях это может привести к недопустимым издержкам или чрезмерному усложнению класса.

☺ Нужно инкапсулировать вычисление таким образом, чтобы его клиенты ничего не знали о том, является ли это вычисление асинхронным или синхронным.

РЕШЕНИЕ

Вместо того чтобы заставить метод прямо возвращать результат вычисления, вынуждаем его возвращать объект, который инкапсулирует это вычисление. Этот объект будет содержать метод, вызываемый в том случае, когда нужно будет получить результат вычисления. Вызывающие этот метод не знают, выполняется ли вычисление синхронно или асинхронно по отношению к их потоку.

На рис. 9.38 показаны классы в этом решении. Опишем роли, выполняемые классами в шаблоне Future.

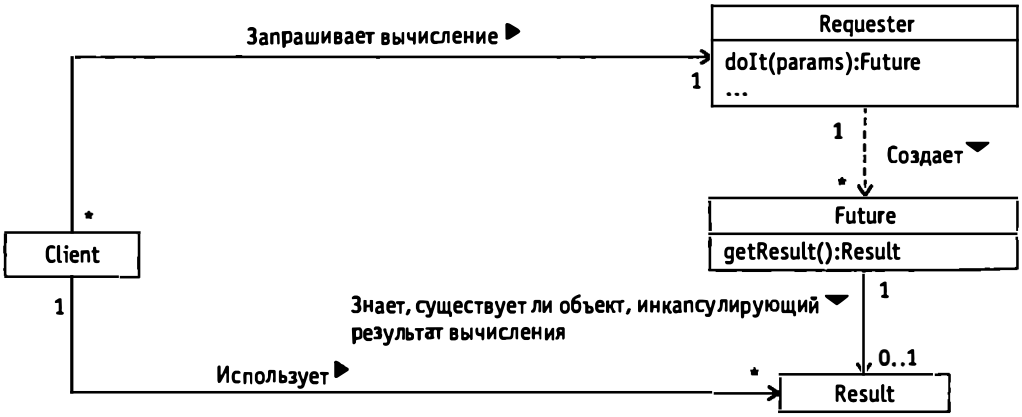


Рис. 9.38. Шаблон Future

Result. Экземпляры класса в этой роли инкапсулируют результат вычисления.

Client. Экземпляры классов в этой роли используют объекты `Result`. Объекты `Client` инициируют процесс получения объекта `Result`, передавая соответствующие параметры методу `doIt` объекта `Requester`.

Requester. Класс в этой роли имеет метод, вызываемый для инициирования вычисления объекта `Result`. На рис 9.38 этот метод показан как метод `doIt` . Метод `doIt` возвращает объект `Future`, который инкапсулирует это вычисление.

Future. Каждый экземпляр класса в этой роли инкапсулирует вычисление объекта `Result`. Это вычисление может быть синхронным или асинхронным. Если

вычисление асинхронное, оно выполняется в своем собственном потоке хронное вычисление может быть уже на стадии выполнения, когда со инкапсулирующий его объект `Future`.

Объекты `Future` содержат метод, который вызывается объектами `Cli` получения результата вычисления. На рис. 9.38 этот метод указан как `get` Способ работы этого метода зависит от того, синхронным или асинх является вычисление.

Если вычисление асинхронное и этот метод вызывается до окончания вы ния, то этот метод ожидает завершения вычисления и затем возвращает тат. Если метод вызывается после окончания вычисления, то он сразу щает результат.

Если вычисление синхронное, при первом вызове метода вычисление сн но завершается и возвращается его результат. Последующие обращения методу просто возвращают результат. С другой стороны, вычисление мо полняться тогда, когда создается объект `Future`.

На рис. 9.39 представлена диаграмма взаимодействия, демонстрирующа модействия между объектами в рамках шаблона Future. Опишем эти действия.

1. Метод `Client` вызывает метод `doIt` объекта `Requester`. Тот инн вычисление, инкапсулированное в объекте `Future`, возвращаемом м `doIt`.
2. Затратив некоторое время на другие действия, объект `Client` вы метод `getResult` объекта `Future`. Этот метод возвращает резуль числений.

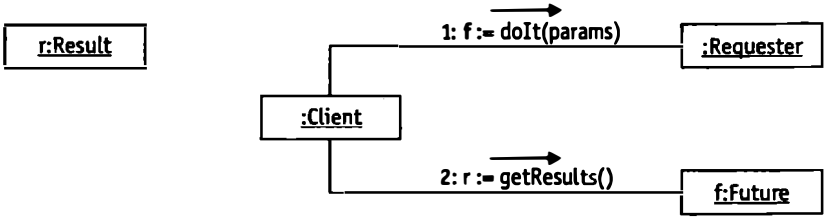


Рис. 9.39. Взаимодействие в шаблоне Future

РЕАЛИЗАЦИЯ

Опрос

Общая схема выполнения шаблона Future предусматривает обращение `Client` к методу `doIt` объекта `Requester`, чтобы начать вычисление, как ему понадобится результат. Затем объект `Client` занят другими дейс

Когда ему наконец потребуется результат вычисления, он вызывает метод `getResult` объекта `Future`. Если вычисление является асинхронным, максимально раннее инициирование вычисления и максимально позднее чтение результата позволяют в наибольшей степени воспользоваться преимуществами асинхронной обработки.

В некоторых ситуациях такая схема не действует. Например, когда у клиента есть активное задание, которое он постоянно выполняет. Рассмотрим пример сервлета, который генерирует Web-страницу, содержащую новости и информацию о погоде. Предположим, что этот сервлет применяет связанные с погодой классы, рассмотренные в разделе «Контекст». После того как он запросил информацию о погоде, нельзя рассматривать ни один момент как самое последнее возможное время получения информации о погоде. Сервлет непрерывно обновляет информацию, используемую им в качестве содержимого Web-страницы. Нужно, чтобы ни один запрос сервлета, предоставляющего содержимое Web-страницы, не был задержан, так как это приведет к задержке обновления информации о погоде.

Один из способов решения этой проблемы предусматривает применение отдельного потока для каждого запроса на получение информации о погоде. Но использование потока для этой цели не слишком выгодно и, скорее всего, приведет к расходованию ресурсов.

Более удачное решение состоит в том, чтобы добавить еще один метод в класс `Future`. Как правило, имя этого метода начинается со слова «check», например, `checkComputation`. Назначение метода `check` в том, чтобы проверять, будет ли объект `Client` ждать результатов вычислений, если он вызовет метод `getResult` объекта `Future`. Метод `check` может быть полезен в том случае, когда инкапсулированное вычисление является асинхронным. Метод `check` возвратит false, если он вызывается до того, как закончилось вычисление, или true, если он вызывается после завершения вычисления. Если вычисление синхронное, метод `check` всегда возвращает true.

Использование метода `check` позволяет объектам `Client` опрашивать объекты `Future`. Каждый запрос на получение содержимого может опрашивать объект `Future`, вызывая его метод `check`. Если метод `check` возвращает true, сервлет обновляет информацию о погоде, которая используется Web-страницей. Если метод `check` возвращает false, то сервлет продолжает использовать ту информацию, которая уже имеется.

Заместитель

Шаблон Future иногда используется вместе с шаблоном Proxy. Обычно это делается так: класс, выступающий в роли `Future`, реализует интерфейс, который может использоваться как заместитель для класса, выполняющего вычисление. При таком подходе объект, инициирующий вычисление и создание объекта `Future`, и объект, который получает результат от объекта `Future`, — это, как правило, разные объекты.

Некоторый объект будет инициировать создание объекта Future. Он будет передавать объект Future другим объектам, которые будут получать доступ к этому объекту через интерфейс, ничего не зная о классе объекта Future или о его участии в шаблоне Future. Когда шаблоны Future и Proxy объединяются подобным образом и вычисление является синхронным, результатом будет шаблон Virtual Proxy.

Запуск синхронного вычисления

Если объект Future инкапсулирует синхронное вычисление, он должен иметь возможность запустить вычисление при первом вызове его метода getResult. Чтобы запустить вычисление, объект Future должен иметь возможность получить все необходимое для вычисления параметров. Поэтому, если класс Future инкапсулирует синхронное вычисление, конструктор этого класса должен будет получить соответствующие параметры с целью задания значений этих параметров для выполнения вычисления.

Рандеву

Реализация класса Future, работающего с асинхронным вычислением, включает взаимодействие потока, который вызывает метод getResult этого класса с целью ожидания завершения вычисления. Существует название для некоторого механизма синхронизации нескольких потоков с целью ожидания того момента, пока один или некоторые из них не достигнут определенной точки. Название этого механизма — *рандеву*.

Класс java.lang.Thread содержит метод под названием join, который обеспечивает подходящий способ реализации рандеву. Метод join объекта Thread не возвращается до тех пор, пока поток не «умрет». Если поток выполняет асинхронное вычисление, связанное с объектом Future, а затем завершается, то объект Future может вызвать метод join этого потока для взаимодействия с ним.

Поток не обязательно должен «умирать» по окончании вычисления. Он может сделать еще что-то. Он может управляться пулом потоков (шаблон Thread Pool, описанный в книге [Grand2001]), который может многократно использовать этот поток для других вычислений. Чтобы реализовать рандеву таким способом, который не знает, что будет делать поток после окончания асинхронного вычисления, нужно использовать методы wait и notifyAll. Листинг класса AsynchronousFuture в разделе «Пример кода» служит примером применения такого способа.

Исключения

Если вычисление, связанное с объектом Future, генерирует исключение, нужно, чтобы оно могло быть получено методом, вызвавшим метод getResult объекта Future. Если вычисление выполняется синхронно, то любые генерируемые им исключения, естественно, могут быть получены вне рамок метода getResult объекта Future.

Если вычисление является асинхронным и генерируется исключение, то естественный ход событий таков, что исключение будет генерироваться в текущем активном потоке. Чтобы такое исключение могло быть получено методом, вызвавшим метод `getResult` объекта `Future`, нужно перехватить это исключение и задать ссылку на него в одной из переменных экземпляра объекта `Future`. Далее можно реализовать метод `getResult` так, чтобы этот метод генерировал исключение, на которое ссылается переменная экземпляра (если она не равна `null`).

СЛЕДСТВИЯ

☺ Классы, использующие вычисление, освобождаются от какой бы то ни было ответственности за поддержку параллельности.

☺ Если вычисление асинхронное, за все детали синхронизации отвечает класс `Future`.

ПРИМЕНЕНИЕ В JAVA API

Класс `java.awt.MediaTracker` может служить примером шаблона Future в Java API. Класс `MediaTracker` инкапсулирует процесс асинхронной загрузки данных изображения в объект `ImageProducer`, связанный с объектом `Image`. Он исполняет и роль `Requester`, и роль `Future`.

Класс `MediaTracker` не инкапсулирует результат загрузки данных изображения. Он просто инкапсулирует процесс загрузки данных. Объект `ImageProducer` выполняет роль объекта `Result`.

ПРИМЕР КОДА

Пример кода для этого шаблона основан на проектировании классов, предназначенных для считывания текущей информации о погоде, рассматривавшейся в разделе «Контекст». Первым представлен класс `WeatherRequester`, который исполняет роль запрашивающей стороны.

```
public class WeatherRequester {
   /**
    * Объект, который выполняет реальную работу
    * по считыванию информации о погоде.
    */
   WeatherFetchIF fetcher;

   public WeatherRequester(WeatherFetchIF fetcher) {
      this.fetcher = fetcher;
   } // constructor(WeatherFetchIF)
```

```
    /**
     * Инициирует процесс получения текущих данных о погоде
     * для места с заданными географическими координатами.
     */
    public synchronized WeatherFuture getWeather(Coordinate
    location) {
        return new WeatherFuture(fetcher, location);
    } // getWeather(Coordinate)
} // class WeatherRequester
```

Конструктор класса `WeatherRequester` создает объект, который делает ре ную работу по считыванию текущей информации о погоде для данной мес сти. Его метод `getWeather` создает объект `WeatherFuture`, передавая его структору объект, предназначенный для реального получения информ о погоде, и объект, описывающий местность, для которой надо получить формацию о погоде. Класс `WeatherFuture` исполняет роль `Future`.

```
public class WeatherFuture {
    /**
     * Когда координата местности передается конструктору,
     * он помещает эту координату, предназначенную для запроса,
     * в данную переменную экземпляра так, чтобы ее видел новый
     * запускаемый поток.
     */
    private Coordinate location;

    /**
     * Объект, используемый для считывания текущей информации
     * о погоде.
     */
    private WeatherFetchIF fetcher ;

    /**
     * Объект, инкапсулирующий логику процесса получения
     * информации.
     */
    private AsynchronousFuture futureSupport ;

    /**
     * Создает объект WeatherFuture, который инкапсулирует
     * получение информации о погоде для данной координаты
     * местности, используя заданный объект WeatherFetchIF.
     */
```

```
  public WeatherFuture(WeatherFetchIF fetcher,
            Coordinate location) {
    this.fetcher = fetcher;
    this.location = location;
    futureSupport = new AsynchronousFuture();
    new Runner().start();
  } // constructor(WeatherFetchIF, Coordinate)

  /**
    * Возвращает true, если запрошенная информация
    * о погоде была прочитана.
    */
  public boolean check() {
    return futureSupport.checkResult();
  } // checkResult()

  /**
   * Возвращает объект Result для этого объекта Future.
   * Если он еще не получен, ждет, пока он не будет получен.
   */
  public synchronized WeatherIF waitForWeather() throws
                                    Exception {
    return (WeatherIF)futureSupport.getResult();
  } // getResult()

  /**
   * Этот закрытый класс представляет общую логику
   * для асинхронного считывания текущей информации о погоде.
   */
  private class Runner extends Thread {
    public void run() {
      try {
        WeatherIF info = fetcher.fetchWeather(location);
        futureSupport.setResult(info);
      } catch (Exception e) {
        futureSupport.setException(e);
      } // try
    } // run()
  } // class runner
} // class WeatherFuture
```

Если информация о погоде прочитана, она инкапсулируется в экзем
класса, который реализует интерфейс под названием WeatherIF. Логика
вления асинхронным вычислением содержится в многократно использ
классе под названием AsynchronousFuture.

```
public class AsynchronousFuture {
  private Object result;
  private boolean resultIsSet;
  private Exception problem;

  /**
   * Возвращает true, если результат был получен.
   */
  public boolean checkResult() {
    return resultIsSet;
  } // checkResult()

  /**
   * Возвращает объект Result для этого объекта Future.
   * Если он еще не получен, ждет, пока он не будет получен.
   */
  public synchronized Object getResult() throws Exception {
    while (!resultIsSet) {
      wait();
    } // while
    if (problem!=null) {
      throw problem;
    } // if problem
    return result;
  } // getResult()

  /**
   * Обращаемся к этому методу для задания результата
   * вычисления.
   * Этот метод должен вызываться только один раз.
   */
  public synchronized void setResult(Object result) {
    if (resultIsSet) {
      String msg = "Result is already set";
      throw new IllegalStateException(msg);
```

```
        this.result = result;
        resultIsSet = true;
        notifyAll();
    } // setResult(Object)

    /**
     * Если асинхронное вычисление, связанное с этим объектом,
     * генерирует исключение, передаем исключение этому методу,
     * и исключение будет снова сгенерировано методом getResult.
     */
    public synchronized void setException(Exception e) {
        problem = e;
        resultIsSet = true;
        notifyAll();
    } // setException(Exception)
} // class AsynchronousFuture
```

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СВЯЗАННЫЕ С ШАБЛОНОМ FUTURE

Asynchronous Processing. Шаблон Asynchronous Processing позволяет делать вычисление асинхронным. Он часто используется вместе с шаблоном Future.

Observer. Использование шаблона Observer представляет собой альтернативный вариант технологии реализации опроса в объекте Future для определения, было ли завершено связанное с ним асинхронное вычисление.

Proxy. Шаблон Future может объединяться с шаблоном Proxy с тем, чтобы объект Future являлся также заместителем для объекта, который выполняет основное вычисление.

Virtual Proxy. Если шаблон Future объединяется с шаблоном Proxy и вычисление синхронное, то получается шаблон Virtual Proxy.

Active Object. Шаблон Active Object, представленный в [SSRB00], описывает способ объединения шаблона Future с шаблоном Asynchronous Processing.

СПИСОК ЛИТЕРАТУРЫ

[Appleton97] Brad Appleton. «Patterns and Software: Essential Concepts and Termi nology.» www.enteract.com/bradapp/docs/patterns-intro.html.

[ASU86] Alfred V. Aho, Ravi Seti, and Jeffery D. Ullman. *Compilers, Principles, Tech niques and Tools.* Reading, Mass.: Addison-Wesley, 1986.

[Bentley86] Jon Louis Bentley. *Programming Pearls.* New York: ACM, 1986.

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Pat terns.* Chichester, England: John Wiley & Sons, 1996.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Desig Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addi son-Wesley, 1995.

[Grand99] Grand Mark. *Patterns in Java: A Catalog of Reusable Design Patterns Illus trated with UML, Volume 2.* Wiley, 1999.

[Grand2001] Grand Mark. *Java Enterprise Design Patterns: Patterns in Java, Volume 3* Wiley, 2001.

[Larman98] Craig Larman. *Applying UML and Patterns.* Upper Saddle River, N.J. Prentice Hall PTR, 1998.

[Lea97] Doug Lea. *Concurrent Programming in Java.* Reading, Mass.: Addison-Wesley 1997.

[LL01] Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engi neering: Practical Software Development Using UML and Java.* New York: McGraw-Hill 2001.

[Rhiel2000] Dirk Rhiel. *Fundamental Class Patterns in Java.* Unpublished manuscrip (in 2000). www.riehle.org/papers/2000/plop-2000-class-patterns.html.

[Ritchie84] D. Ritchie. «A Stream Input-Output System,» *AT&T Bell Labs Technic Journal*, vol. 63, pp. 311–324. October 1984.

[SSRB00] Douglas Schmidt, Micheal Stal, Hans Rohnert, and Frank Buschmann *Pattern Oriented Software Architecture, Volume 2.* Chichester, England: John Wiley & Sons, 2000.

[Woolf97] Bobby Woolf. «The Null Object Pattern.» www.ksccary.com/nullobj.htm

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

D

N

O

P

T

U

V

W

А

Производственно-практическое издание

Гранд Марк

Шаблоны проектирования в Java

Ведущий редактор *А.В. Жвалевский*
Научный редактор *Д.А. Народецкий*
Редактор *Е.С. Каляева*
Художник обложки *С.В. Ковалевский*
Компьютерная верстка *С.И. Лученок*
Корректор *Е.О. Кликунова*

Подписано в печать с готовых диапозитивов 22.01.2004.
Формат 70×100 1/16. Бумага газетная. Гарнитура Ньютон.
Печать офсетная. Усл. печ. л. 45,37. Уч.-изд. л. 36,87.
Тираж 3000 экз. Заказ № 3072

ООО «Новое знание». ИД № 05902 от 24.09.2001.
107076, Москва, Колодезный пер., д. 2а.
Телефон (095) 234-58-53.
E-mail: ru@wnk.biz
http://wnk.biz

Отпечатано с готовых диапозитивов
в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12