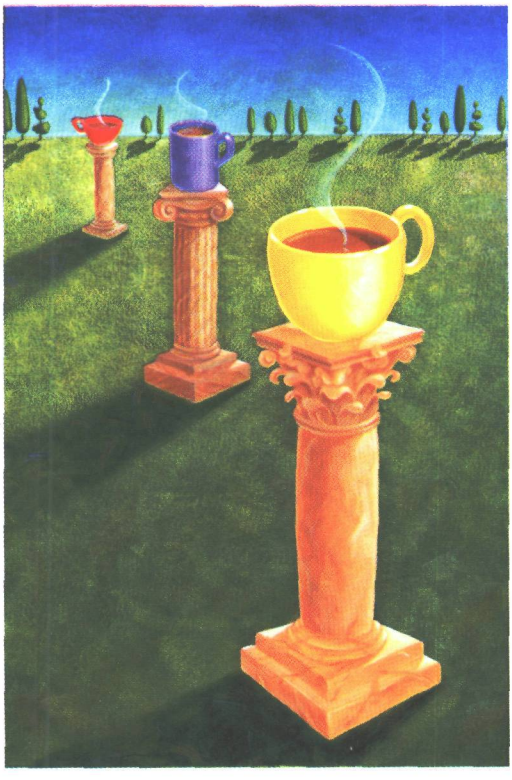
ПРИМЕНЕНИЕ ШАБЛОНОВ JAVA



- Типовые решения для всех этапов разработки приложения
- Рассмотрено 32 шаблона, в том числе 23 шаблона, знание которых обязательно для сдачи экзамена на сертификат Sun Certified Enterprise Architect
- Описание использования шаблонов в API Java, включая API динамической подгрузки классов во время выполнения, обеспечения безопасности, AWT/Swing, RMI, JDBC™, J2EE™ и др.

СЕРИЯ JAVA
от Sun Microsystems Press

Стивен Стелтинг • Олав Маассен

ПРИМЕНЕНИЕ ШАБЛОНОВ JAVA™

Библиотека профессиоиала

APPLIED JAVA™ PATTERNS

ПРИМЕНЕНИЕ ШАБЛОНОВ JAVA™

Библиотека профессионала

СТИВЕН СТЕЛТИНГ
ОЛАВ МААССЕН

Москва • Санкт-Петербург • Киев
2002

ББК 32.973.26-018.2.75
С79
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского под редакцией *А.А. Чекаткова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Стелтинг, Стивен, **Маасен,** Олав.

С79 Применение шаблонов Java. Библиотека профессионала. : Пер. с англ. — М.: Издательский дом "Вильяме", 2002. — 576 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0339-4 (рус.)

Эксперты компании Sun Microsystems Стив Стелтинг и Олав Маассен создали практическое руководство, содержащее описание проверенных временем методов использования всех типов шаблонов, представляющих как собой целые архитектуры систем, так и отдельные простые классы применительно к платформе Java. В начале книги приведена история возникновения и развития шаблонов проектирования, а также рассматриваются методы эффективного применения этих шаблонов. Далее следует каталог шаблонов, сгруппированных по основным категориям: производящие, поведенческие, структурные и системные. Кроме того, авторы описывают шаблоны и соответствующие приемы их использования для основных API как языка Java, так и API, используемых при разработке распределенных приложений.

Книга будет полезной как начинающим, так и опытным программистам Java, осваивающим методику использования шаблонов проектирования, а экспертам в этой области может служить в качестве справочника.

ББК **32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Prentice Hall, Ptr.

Authorized translation from the English language edition published by Prentice Hall PTR, Copyright © 2002 Sun Microsystems, Inc.

All rights reserved. No part of this hook may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2002

ISBN 5-8459-0339-4 (рус.)
ISBN 0-13-066190-2 (англ.)

© Издательский дом "Вильяме", 2002
© Prentice Hall, Inc., 2001

Оглавление

Содержание

Моим родителям, близким и друзьям: спасибо вам за все. Жизнь прекрасна благодаря вам.

Стив Стелтинг.

Памяти Джоири Хаттерс. Значение жизни человека определяется тем, что он сделал для других.

Олав Маассен.

ПРЕДИСЛОВИЕ

Почему мы написали эту книгу

За время своей преподавательской практики мы неоднократно сталкивались с тем, что большинство программистов-практиков, изучающих язык программирования Java, не могут четко объяснить, что такое шаблоны проектирования. Только один из десяти слушателей может припомнить название нескольких самых популярных шаблонов. С другой стороны, концепции, на которых основаны те или иные шаблоны проектирования, хорошо знакомы многим программистам, поскольку стоило нам продемонстрировать эти шаблоны на занятиях, как их тут же узнавали многие наши слушатели.

Поэтому мы решили создать нечто вроде каталога шаблонов проектирования для разработчиков, использующих язык программирования Java. Этот каталог, по нашему мнению, должен помочь, прежде всего тем разработчикам, которые хорошо понимают, какие преимущества дает применение шаблонов, и хотели бы работать с ними, но нуждаются в практическом руководстве, объясняющем, как и в каких случаях нужно применять тот или иной шаблон. Исходя из этого мы постарались написать книгу простым и понятным языком, а также привели полный текст всех рассмотренных в ней примеров, позволяющий получить полноценные работающие программы.

Мы будем считать, что справились с этой задачей, если вы, прочитав эту книгу, не только ознакомитесь с шаблонами проектирования и языком программирования Java, но и получите от нее удовольствие.

О чем эта книга

Эта книга призвана дать вам основные сведения, необходимые для практического применения шаблонов проектирования при разработке реальных Java-приложений. Кроме того, в этой книге показано, в каких случаях и почему следует отдавать предпочтение именно шаблонам проектирования, а не другим средствам API языка Java.

Для кого предназначена эта книга

Данная книга была написана прежде всего для опытных Java-программистов, которые желают изучить прогрессивные методы разработки приложений. Поэтому, приступая к чтению, вы должны хорошо знать язык программирования Java и иметь хотя бы базовые понятия об основных прикладных программных интерфейсах (API) этого языка. Также вам будут полезны знания базовых понятий UML, хотя для работы

с книгой эти знания не являются обязательными. Мы можем порекомендовать книги Мартина Фаулера и Кендалла Скотта "UML в кратком изложении. Применение стандартного языка объектного моделирования" (М., Издательский дом научно-технической литературы "Мир", 1999. с. 192) или Крэга Лармана "Применение UML и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование" (М., Издательский дом "Вильямс", 2001. с. 496).

Соглашения, используемые в данной книге

На страницах данной книги все листинги и фрагменты исходного текста программ набраны моноширинным шрифтом. Этот же шрифт используется и в тексте, когда обсуждаются те или иные классы, интерфейсы, методы или переменные. При этом идентификаторы вида methodName применяются для обозначения всех методов с таким именем, тогда как идентификатор вида `methodName ()` — для обозначения какого-то конкретного метода, у которого нет параметров.

Имена абстрактных классов начинаются словом Abstract, а классов, которые реализуют интерфейс или образуют подкласс, — словом Concrete (конечно, кроме тех случаев, когда подкласс также является абстрактным). Все эти соглашения приведены на схеме, представленной на рис. 1.

Термином *клиент* (client) мы будем обозначать любой класс, который использует классы того или иного шаблона проектирования. Мы ввели это понятие, чтобы не путать клиентов с пользователями. Термин *пользователь* (user) мы применяем для обозначения человека, работающего с приложением.

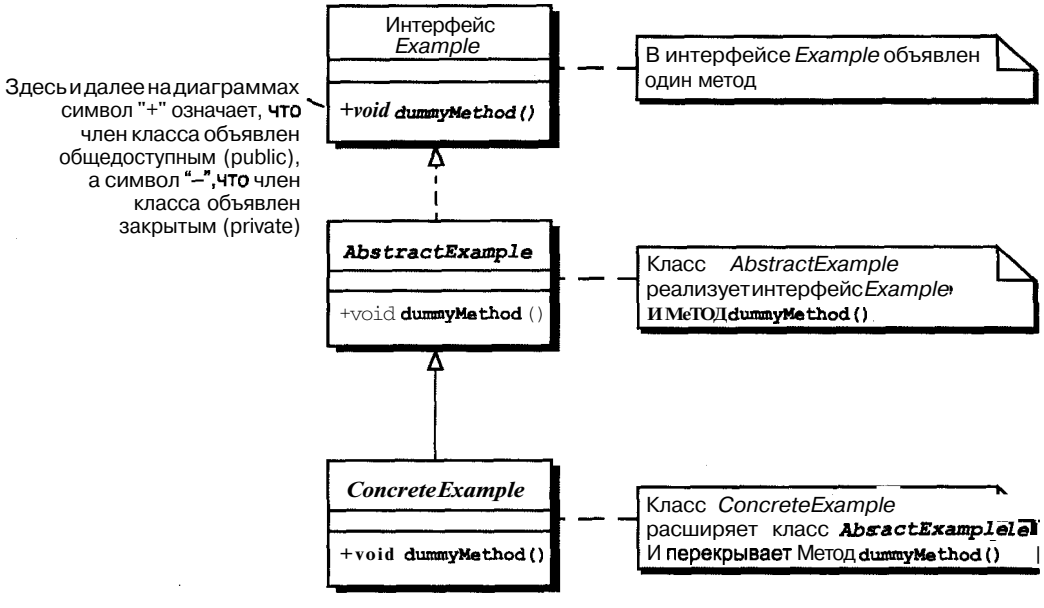


***Рис.** 1.* Обозначения, используемые в данной книге

Аббревиатуры вида [CJ2EEP], которые можно встретить в разделах "Родственные шаблоны", означают ссылки на другие источники информации по использованию шаблонов. Полный перечень этих источников приведен в конце данной книги в приложении А.

Структура книги

Книга состоит из двух частей. Часть I, "Общеупотребительные шаблоны", по структуре подобна справочнику или каталогу шаблонов.

Глава 1, "Производящие шаблоны", посвящена описанию шаблонов, применяемых для создания объектов: Abstract Factory, Builder, Factory Method, Prototype и Singleton.

В главе 2, "Поведенческие шаблоны", рассматриваются шаблоны, которые позволяют определить поведение разрабатываемой объектной модели: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method и Visitor.

В главе 3, "Структурные шаблоны", описаны шаблоны, с помощью которых можно структурировать приложения: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, HOPP и Proxy.

В главе 4, "Системные шаблоны", представлены шаблоны, помогающие получить улучшенные архитектурные решения: Callback, Router, MVC, Session, Successive Update, Transaction и Worker Thread.

Часть II, "Шаблоны в языке программирования Java", содержит основные сведения о программных интерфейсах (API) Java (глава 5), а также о том, какие преимущества дает программисту использование шаблонов, рассматриваемых API.

В главе 6, "Основные API языка Java", приведен обзор основных API, таких как API обработки событий, JavaBeans, AWT, Swing, коллекций, ввода/вывода и динамической подгрузки классов во время выполнения.

Глава 7, "Технологии распределенной обработки данных", посвящена описанию некоторых API распределенной обработки данных и особенностям применения шаблонов таких API, как JNDI, JDBC, RMI и CORBA.

Глава 8, "Архитектуры Jini и J2EE", содержит информацию о двух дополнительных библиотеках Jini и J2EE. В частности, в данной главе рассмотрены такие технологии J2EE, как сервлеты, JSP и EJB.

Как пользоваться этой книгой

Можно, конечно, прочесть эту книгу от начала до конца, не пропуская ни одной страницы. Однако мы рекомендуем начать с простых шаблонов (Factory Method, Singleton, Observer и Composite), а затем уже продолжить изучение остального материала. Другой подход заключается в том, чтобы сразу перейти к чтению материала второй части, если какой-то из описанных в ней API хорошо вам знаком, чтобы на примере этого API быстрее понять, как нужно использовать шаблоны.

Описания шаблонов можно изучать в любом порядке. Мы постарались расположить материал так, чтобы впоследствии вы могли обращаться к нему, как к справочнику, когда у вас возникнет необходимость освежить знания в процессе применения шаблонов на практике.

Web-узел, посвященный данной книге

Специально для этой книги мы разработали Web-узел, на котором размещаем обновленную информацию и другие материалы. Адрес узла: `http://www.phptr.com/appliedjavapatterns`.

Благодарности

Любая книга представляет собой, конечно же, плод коллективных усилий. Мы хотели бы поблагодарить всех, кто воплотил эту книгу в жизнь. Нам посчастливилось работать с чудесными людьми, и мы решили посвятить эту страницу им, чтобы они знали, как мы ценим их вклад в общее дело.

Грэгу Дончу (Greg Doench), провидцу из издательства Prentice Hall. Спасибо вам за то, что вы были "великим кормчим" этого проекта. Приступив к работе, мы выяснили, что Грэг увлекается марафоном. Когда Стив намекнул, что он хотел бы попробовать свои силы в этом виде спорта, Грэг ответил: "Не раньше, чем закончим книгу". Теперь мы понимаем, почему он так сказал: написание книги очень похоже на марафонский забег. Примите нашу искреннюю благодарность за вашу постоянную помощь и поддержку и за вашу веру в успех этого проекта.

Рашель Борден (Rachel Borden), светилу Sun Press. Спасибо вам за руководство проектом на всем его протяжении, вплоть до публикации книги. Если бы не вы, мы бы растеряли все свои идеи в тех листках, которыми были облеплены наши рабочие места. Спасибо вам за постоянную поддержку и внимание, а также за терпение, которое вы проявляли, растолковывая технарям тонкости издательского дела. Отдельная благодарность за то, что вы нередко вставали спозаранку, чтобы провести телеконференцию с людьми, находящимися на другой стороне Земного шара. Но больше всего мы благодарим вас за то, что в нашей работе вы всегда были лидером, задающим темп.

Сольвейг Хогланд (Solveig Haugland), выдающемуся редактору. Спасибо вам за то, что поверили в мечту, и за то, что помогли превратить ее в реальность. Спасибо вам за то, что шли до конца и превратили груду бессвязных идей в нечто гораздо более значимое. Наконец, спасибо вам за то, что показали нам, что даже в технической книге можно найти место для юмора.

Нашим талантливым техническим рецензентам. Спасибо за то, что заставили нас как следует думать о том, что мы намереваемся сказать. Наша самая искренняя благодарность Дженни Йип (Jennie Yip) за потраченное ею время, когда она тщательно описывала все найденные в тексте неточности, а также Брайну Бэшаму (Brayan Basham), Берту Бейтсу (Bert Bates), Джону Крапи (John Crupi), Джиму Голлентайну (Jim Gallentine), Вернеру ван Муку (Werner van Mook), Нанно Скерингу (Nanno Schering), Юргену Скимбера (Juergen Schimbera), Роберту Скрайджверсу (Robert Schrijvers) и Фреду Зюйдендорпу (Zuijdendorp).

Выражаем сердечную благодарность сотрудникам издательства Prentice Hall. Мы искренне просим извинить нас за то, что не смогли встретиться со всеми вами лично, но знаем, что именно вы превращаете идеи авторов в реальные книги. Ваша работа поистине прекрасна — вы помогаете принести в мир мечты и идеи. Мы благодарим вас за вашу преданность работе как над этой, так и над другими книгами (многие из которых заняли подобающее им место и в наших сердцах, и на наших книжных полках).

Стефен Стелтинг хотел бы поблагодарить Стива Брэдшоу (Steve Bradshaw), Аннет Балденегро (Annette Baldenegro), Синди Льюис (Cindy Lewis) и других менеджеров Sun Educational Services. Спасибо за вашу поддержку и за веру в нас, которые вы демонстрировали на протяжении последнего года. Ваши помощь и понимание так дороги для нас, что слишком трудно выразить это словами.

Олаф Маассен благодарит своих менеджеров Гарри Поландта (Harry Pallandt) и Андрэ Арнольдаса (Andre Arnoldus) за возможность часто оставаться на работе по окончании рабочего дня, а также за поддержку на протяжении многих лет.

Ингрид и Нильс — вы две самые большие звезды в моей вселенной. Спасибо вам за поддержку и вдохновение, благодаря которым я закончил этот проект.

Брит — ты моя третья "звездочка". Благодарю тебя за то, что не спешил появляться на свет, пока я не закончу книгу. Теперь я могу всецело заняться следующим большим проектом — моей семьей.

ВВЕДЕНИЕ

Почему шаблоны?

Если бы строители строили дома так, как программисты пишут программный код, достаточно было бы одного-единственного дятла, чтобы разрушить всю цивилизацию.

Если бы вы решили построить дом, как бы вы это делали?

Можно к этой работе подойти так, как некоторые и поступают, когда собираются построить скворечник.

1. Найти подходящее дерево.
2. Приготовить доски, молоток и гвозди.
3. Взобраться на дерево, применить материалы и инструменты, перечисленные в п. 2.
4. Спуститься и надеяться, что все обойдется.

Конечно, любой, кто хоть раз пытался действовать подобным образом, знает, что результат, скорее всего, будет плачевным — в некоторых случаях гибнет не только скворечник, но и дерево. Более правильный подход состоит в том, чтобы найти мастера и попросить его помочь в подготовке чертежей. Если же вы собираетесь построить не скворечник, а настоящий дом, большинство здравомыслящих людей обратиться за помощью к архитектору.

Однако на основе чего архитектор, будучи экспертом по строительству домов, принимает те или иные решения? Возможно ли, располагая многолетним опытом, применить его для строительства совершенно нового, нетипичного дома? По-видимому, есть нечто, основанное на знаниях, опыте и, возможно, в некоторой степени на интуиции, что позволяет архитектору достичь успеха.

Вопросы, связанные со строительством и проектированием зданий, в действительности не очень отличаются от тех, с которыми мы сталкиваемся в мире разработки программного обеспечения (ПО). Каким образом спроектировать по-настоящему хорошую программу? Как применить опыт прошлых лет к проектам, нацеленным на будущее? На основе чего нужно принимать решения при проектировании программы, которые позволили бы получить продукт, обладающий хорошими показателями, такими как гибкость, расширяемость и эффективность?

Как и при строительстве зданий, нам нужен опытный руководитель, который будет играть роль архитектора. Знания, опыт и здравый смысл так же нужны при про-

ектировании ПО, как и при проектировании домов. Иными словами, нам нужен гуру в области разработки ПО.

Однако в мире не так уж много гуру. И до тех пор, пока технология клонирования не получила всеобщего одобрения, нам чаще всего придется рассчитывать на свои силы. Нам придется растить экспертов по разработке ПО в ходе выполнения своих проектов, отбирая потенциальных кандидатов среди сотрудников своих компаний.

Итак, все вернулось на круги своя. Мы хотим спроектировать хорошее ПО, но не знаем при этом, какие архитектурные решения будут правильными, позволяющими нам получить качественный продукт. Мы хотим вырастить опытных разработчиков ПО, но даже не представляем, как собрать воедино знания об эффективном проектировании, накопленные современными разработчиками ПО.

Может быть, существует способ собрать эти знания? Может быть, мы можем приобщиться к опыту гуру? Как нам записать и обобщить ключевые концепции проектирования ПО, положив тем самым фундамент для нового поколения разработчиков?

Да, решение проблемы существует и имя ему "шаблоны проектирования".

Они хорошо документированы, поэтому эксперты часто без труда справляются с новыми проблемами, просто применяя решения, которые уже работали в подобных ситуациях в прошлом. Сначала эксперты вычленяют составные части проблемы, требующей решения, которые подобны задачам, возникавшим перед ними в прошлом. Затем они вспоминают, какие решения применялись в прошлый раз, и обобщают эти решения. Наконец, эксперты адаптируют общее решение к контексту частной проблемы.

Идея, на которой основывается применение шаблонов проектирования, заключается в том, чтобы выработать стандартизованный подход к представлению общих решений, пригодных для часто встречающихся ситуаций при разработке ПО. Такой подход имеет ряд преимуществ.

- Со временем можно получить каталог шаблонов. Это позволяет новичкам в разработке ПО более эффективно использовать многолетний опыт их предшественников.
- Все решения, принимаемые при проектировании ПО, снабжены формализованным описанием, позволяющим оценить как достоинства, так и недостатки того или иного решения. Стандартизованные шаблоны облегчают как для новичков, так и для экспертов в области разработки ПО понимание того, как влияют на архитектуру создаваемой программы те или иные решения.
- Шаблоны проектирования позволяют всем, кто их использует, "говорить на одном языке". Это, в свою очередь, облегчает взаимопонимание между разработчиками при выборе того или иного решения. Вместо того чтобы детально описывать какое-либо архитектурное решение, мы говорим, какой шаблон мы намерены использовать.
- Мы можем соотносить шаблоны друг с другом, что позволяет разработчику увидеть, какие шаблоны могут использоваться в одном проекте.

Шаблоны проектирования предоставляют в наше распоряжение эффективный способ делиться опытом со всеми участниками сообщества объектно-ориентированного программирования. Независимо от того, каким языком программирования мы владеем (C++, Smalltalk или Java), и того, в какой области мы приобрели опыт проектирования ПО (Web-проекты, интеграция устаревших систем или заказной проект),

мы можем накапливать свой собственный опыт и делиться им с другими разработчиками. Если же говорить о долгосрочной перспективе, то данный подход позволяет улучшить состояние дел во всей индустрии разработки ПО.

Историяприменениешаблонов

Они прибыли из иных миров... из Университета Калифорнии в Беркли

Идейным отцом применения шаблонов проектирования при разработке ПО считают профессора архитектуры Университета Калифорнии в Беркли Кристофера Ллегсандера (ChristoferAlexander). В конце 70-х годов прошлого века он опубликовал несколько книг, в которых изложил основные принципы применения шаблонов в архитектуре, а также поместил каталог архитектурных шаблонов.

Именно посвященные шаблонам работы Алегсандера и привлекли внимание к проблеме программистов, интересующихся объектно-ориентированным программированием (ООП). В их среде появились пионеры применения шаблонов в разработке ПО, которые на протяжении последующих десяти лет сформулировали основные принципы этого метода. Среди первопроходцев были Кент Бэк (Kent Beck) и Уард Каннингхэм (Ward Cunningham). В 1987 году на конференции OOPSIA (Object-Oriented Programming, Systems, and Applications), посвященной ООП, прозвучал их доклад о применении шаблонов проектирования в языке Smalltalk. Еще одним адептом нового подхода стал Джеймс Коплайен (James Coplien), который в начале 90-х гг написал книгу о применении идиом (т.е. шаблонов) при разработке ПО на языке C++.

Ежегодные конференции OOPSLA сослужили хорошую службу для роста сторонников технологии шаблонов, так как на ее мероприятиях энтузиасты могли свободно делиться своими идеями со множеством благодарных слушателей. Кроме того, важную роль в становлении данного направления технологии разработки ПО сыграли конференции некоммерческой организации Hillside Group, создателями которой были Кент Бэк и Гради Буч (Grady Booch).

Однако по-настоящему ощутимый вклад в дело популяризации технологии шаблонов проектирования внесла изданная в 1995 году книга *Design Patterns: Elements of Reusable Object-Oriented Software.*[1] Ее авторы Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides) приложили так много усилий для распространения своих идей, что заслужили шутливое прозвище "Четверка" (GoF — gang of four). В книге представлено введение в донольно сложный язык шаблонов с иллюстрациями реализации обсуждаемых шаблонов на языке C++. Среди прочих книг, существенно повлиявших на развитие технологии шаблонов, можно отметить работу Френка Баскмана (Frank Buschmann), Регины Меньe (Regine Meunier), Ханса Ронерта (Hans Rohnert) и Майкла Стала (Michael Stal) под названием *Pattern-Oriented Software Architecture, A System of Patterns.*

Выход в свет этих двух книг о шаблонах проектирования привлек внимание к технологии шаблонов и вызвал интерес в самых широких программистских кругах. В это же время началось бурное развитие технологии Java, поэтому неудивительно, что

---

[1] *Гамма Э., Хелм Р., Джонсон Р., ВлиссидесДж. Приемыобъектно-ориентированного проектирования. Паттерны проектирования. - СПб.: Издательство "Питер", 2001. 368 с.*

Java-разработчики с самого начала стали применять шаблоны проектирования в своих проектах. Рост популярности технологии шаблонов проектирования среди Java-разработчиков проявился как в виде специальных презентаций на конференциях, таких как JavaOne, так и в появлении отдельной рубрики по шаблонам проектирования практически во всех специализированных журналах по программированию на Java.

Основные концепции технологии шаблонов

Поговорим том, о сем...

В центре технологии шаблонов лежит идея стандартизации информации о типичной проблеме и о методах ее решения. Один из самых полезных результатов, полученных в работе Алегсандера, состоит в выделении некоторого способа представления шаблонов, который впоследствии был назван *формой* (form), или *форматом* (format). В форме Алегсандера применялось пять направлений, по которым формализовалось обсуждение шаблонов и их применение в конкретных ситуациях.

Самое главное преимущество такого подхода состоит в том, что даже само название шаблона представляет собой ответ на вопрос: "Что можно сделать с помощью этого шаблона?" Кроме того, в форме содержится: описание проблемной области; пояснение того, как данный шаблон решает означенную проблему; в чем заключаются преимущества, недостатки и, возможно, компромиссы при использовании данного шаблона.

Естественно, когда шаблоны были восприняты миром ООП, появились вариации формы Алегсандера, призванные учитывать интересы разработчиков ПО. Большинство из существующих сегодня форм были построены на одной или двух формах, получивших название "Канонических", или форм "Четверки". В данной книге используется одна из вариаций форм "Четверки", согласно которой при представлении шаблонов освещаются следующие вопросы.

- *Название (Name).* Слово или выражение, описывающее основное назначение шаблона.
- *Также известен как (Also Known As).* Альтернативные названия (если, конечно, таковые имеются).
- *Свойства шаблона (Pattern Properties).* Классификация шаблона. Мы будем определять шаблон, используя термины из двух следующих групп.

  Тип:

  - *Производящие шаблоны (Creational patterns),* предназначенные для создания объектов.
  - *Поведенческие шаблоны (Behavioral patterns),* обеспечивающие координацию функционального взаимодействия между объектами.
  - *Структурные шаблоны (Structural patterns),* используемые для управления статическими, структурными связями между объектами.
  - *Системные шаблоны (System patterns),* предназначенные для управления взаимодействием на системном уровне.

Уровень:

- *Отдельный класс (single class).* Шаблон применяется к отдельному, независимому классу.
- *Компонент (component).* Шаблон используется для создания группы классов.
- *Архитектурный (architectural).* Шаблон используется для координации работы систем и подсистем.

- *Назначение (Purpose).* Короткое описание назначения шаблона.
- *Представление (Introduction).* Описание типичной проблемы, для решения которой можно использовать данный шаблон, подкрепленное примером.
- *Область применения (Applicability).* Определение ситуаций, в которых можно использовать данный шаблон проектирования.
- *Описание (Description).* Более подробное обсуждение логики работы и поведения шаблона.
- *Реализация (Implementation).* Описание того, что нужно сделать при реализации данного шаблона. Если вы решили, что этот шаблон вам подходит, в данном разделе вы найдете сведения о том, как практически реализовать программный код, построенный на его основе.
- *Достоинства и недостатки (Benefits and Drawbacks).* Что вам даст использование данного шаблона и чем, возможно, придется пожертвовать.
- *Варианты (Pattern Variants).* Возможные варианты реализации и (или) самого шаблона.
- *Родственные шаблоны (Related Patterns).* Описание других шаблонов, которые близки по назначению с рассматриваемым шаблоном или каким-то образом связаны с ним.
- *Пример (Example).* Пример использования шаблона в программе, написанной на языке Java.

Повторное использование и абстракция программного кода

или ...закончу и начну сначала...

Технология шаблонов проектирования стала важным эволюционным этапом в развитии таких концепций, как *абстракция (abstraction)* и *повторное использование (reuse)* программного кода. Обе эти концепции занимают центральное место в самой идее программирования (некоторые даже полагают, что они являются важнейшими концепциями).

Абстракция — это метод, с помощью которого разработчики могут решать сложные проблемы, последовательно разбивая их на более простые. Затем можно использовать имеющиеся готовые решения полученных типовых простых проблем в качестве строительных блоков, из которых разработчики получают решения, пригодные для реализации повседневных сложных проектов.

Повторное использование не менее важно для разработки ПО. Можно без преувеличения сказать, что вся история разработки ПО отмечена постоянным поиском все более и более изощренных методов повторного использования программного кода. В чем же тут дело? Зачем все это нужно? Оказывается, что повторное использование — это, по сути, цель, к которой устремлена технология разработки ПО по своей природе. Представим, например, что нужно создать в условиях жестких временных ограничений крупный программный проект. Какой путь вы бы выбрали?

- Написать весь код "с нуля", заставив и себя, и всех окружающих пройти через длительный и болезненный процесс тестирования и проверки всего написанного кода.
- Использовать проверенный и протестированный код в качестве фундамента своего проекта.

Поймите меня правильно: я прекрасно знаю, что кодирование — это тяжкий труд. Это тестирование, отладка, документирование и послепродажное сопровождение, т.е. все то, что так в большинстве своем не любят разработчики. Поэтому мы и придумали несколько методов повторного использования кода и концепций разработки ПО.

- Раньше других появился метод повторного использования, основанный на технологии "скопируй и вставь", или, проще говоря, на вставке в новые программы фрагментов ранее созданных программ. Об эффективности такого, с позволения сказать, повторного использования, лучше умолчать. Кроме того, данный метод также не дает никаких сколько-нибудь заметных качественных преимуществ с точки зрения абстракции кода.
- Более гибкий метод повторного использования состоит в повторном использовании алгоритмов. В соответствии с этим методом, разработчик может использовать любой однажды разработанный алгоритм, например такой, как алгоритм сортировки или поиска, для того, чтобы абстрагировать подход (обычно, основанный на математических выкладках) к решению вычислительных задач определенного класса.
- Функциональное повторное использование программного кода, а также его "идеологический противник" — повторное использование структур данных, позволяют обеспечить непосредственное повторное использование абстракции кода. Например, любой разработчик, задавшийся целью смоделировать нечто наподобие адреса, может определить структуру, содержащую все необходимые поля, а затем применять повторное использование этой структуры во всех проектах, требующих работы с адресами. Подобным же образом можно определить операцию по начислению налогов в виде функции (процедуры, подпрограммы, метода — в зависимости от выбранного языка программирования) `computeTax`, а затем копировать ее в каждый новый проект.

Двумя расширениями концепции повторного использования кода являются библиотеки функций и API. Они представляют разработчику возможность получить полный пакет функциональности, доступный для всех последующих приложений без необходимости копирования программного кода из приложения в приложение.

В ходе развития объектно-ориентированных языков программирования был совершен огромный скачок вперед в области абстракции и повторного использования программного кода. С помощью этой технологии была создана целая плеяда высокопроизводительных методов его создания.

Например, чего стоит только одна концепция класса как "эталона" объектов, с ее объединением двух ранее возникших механизмов: функциональной абстракции и абстракции данных. Объединяя структуру сущности (данные) с функциональностью, которая применяется к сущности (поведение), мы получаем эффективный метод повторного использования программного элемента.

Помимо основополагающего понятия класса, объектно-ориентированные языки принесли с собой множество других методов использования существующего кода. В качестве примера можно привести такие концепции, как концепции подклассов и интерфейсов, открывших совершенно новые возможности в повторном использовании программного кода при разработке ПО. Наконец, можно вспомнить о таком мощном механизме, как ассоциация друг с другом групп классов, после чего эти группы рассматриваются как единый логический программный компонент, что обеспечивает возможность получения очень мощной с точки повторного использования модели на системном уровне.

В приведенной ниже табл. I в столбце "Повторное использование" показано, как каждый из рассмотренных методов подтверждает целесообразность повторного использования программного кода.

Таблица I. Сравнение различных подходов к повторному использованию и абстракции программного кода

Метод	*Повторное использование*	*Абстракция*	*Универсальность подхода*
Копирование и вставка	Очень плохо	Отсутствует	Очень плохо
Структуры данных	Хорошо	Тип данных	Средне — хорошо
Функциональность	Хорошо	Метод	Средне — хорошо
Типовые блоки кода	Хорошо	Типизируемая операция	Хорошо
Алгоритмы	Хорошо	Формула	Хорошо
Классы	Хорошо	Данные + метод	Хорошо
Библиотеки	Хорошо	Функции	Хорошо — очень хорошо
API	Хорошо	Классы утилит	Хорошо — очень хорошо
Компоненты	Хорошо	Группы классов	Хорошо — очень хорошо
Шаблоны проектирования	Отлично	Решения проблем	Очень хорошо

В столбце "Абстракция" табл. I указаны сущности, для которых выполняется абстракция, а в столбце "Универсальность подхода" показано то, насколько легко применить соответствующий метод, не прибегая к изменениям или переработке кода. Обратите внимание на то, что показатель степени повторного использования очень сильно зависит от эффективности применения того или иного метода на практике. Это понятно, так как любая, даже самая лучшая возможность может быть использована как правильно, так и неправильно.

Одной из самых выдающихся возможностей, предоставляемых шаблонами проектирования, является то, что они позволяют разработчикам более эффективно применять другие методы повторного использования программного кода. Шаблон в определенных ситуациях может, например, использоваться как руководство для эффективного управления наследованием или же для эффективного назначения связей между классами, обеспечивая тем самым решение той или иной проблемы..

Резюме

Шаблоны проектирования — это весьма ценное инструментальное средство в арсенале разработчика ПО, позволяющее существенно повысить эффективность создаваемого кода. В данной книге представлены лишь некоторые из самых популярных шаблонов проектирования, хотя на самом деле их гораздо больше. Добро пожаловать в мир шаблонов проектирования!

Часть I

ОБЩЕУПОТРЕБИТЕЛЬНЫЕ ШАБЛОНЫ

ПРОИЗВОДЯЩИЕ ШАБЛОНЫ

Глава 1

Введение

Производящие шаблоны (*creational patterns*) предназначены для обеспечения выполнения одной из самых распространенных задач в ООП — создания объектов в системе. В ходе работы большинства объектно-ориентированных систем, независимо от уровня их сложности, создается множество экземпляров объектов. Производящие шаблоны облегчают процесс создания объектов, предоставляя разработчику следующие возможности.

- *Единый способ получения экземпляров объектов.* В системе обеспечивается механизм создания объектов без необходимости идентификации определенных типов классов в программном коде.
- *Простота.* Некоторые из шаблонов упрощают процесс создания объектов до такой степени, что полностью избавляют разработчика от необходимости написания большого и сложного программного кода для получения экземпляра объекта.
- *Учет ограничений при создании.* Некоторые шаблоны позволяют при создании объектов налагать ограничения на их тип или количество в соответствии с установленными требованиями системы.

В данной главе будут обсуждаться следующие производящие шаблоны проектирования.

- *Abstract Factory.* Обеспечивает создание семейств взаимосвязанных или зависящих друг от друга объектов без указания их конкретных классов.
- *Builder.* Упрощает создание сложных объектов путем определения класса, предназначенного для построения экземпляров другого класса. Шаблон Builder генерирует только одну сущность. Хотя эта сущность в свою очередь может содержать более одного класса, но один из полученных классов всегда является главным.

- *Factory Method*. Определяет стандартный метод создания объекта, не связанный с вызовом конструктора, оставляя решение о том, какой именно объект создавать, за подклассами.
- *Prototype*. Облегчает динамическое создание путем определения классов, объекты которых могут создавать собственные дубликаты.
- *Singleton*. Обеспечивает наличие в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру.

Два из перечисленных выше шаблона, а именно Abstract Factory и Factory Method, базируются исключительно на концепции определения создания настраиваемых объектов. Подразумевается, что разработчик, применяющий эти шаблоны, при редактирализации системы обеспечит механизм расширения создаваемых классов или интерфейсов. В силу этой особенности данные шаблоны часто объединяются с другими производящими шаблонами.

Abstract Factory

Также известен как Kit, Toolkit

Свойства шаблона

Тип: производящий шаблон
Уровень: компонент

Назначение

Обеспечивает создание семейств взаимосвязанных или зависящих друг от друга объектов без указания их конкретных классов.

Представление

Предположим, что вы планируете создать в приложении, предназначенном для управления личной информацией (PIM — personal information manager), подсистему, которая управляет данными об адресах и телефонах. Планируется, что PIM-приложение будет выполнять функции адресной книги, личного планировщика, диспетчера встреч и контактов, поэтому оно будет интенсивно обращаться к информации об адресах и номерах телефонов.

Можно начать с разработки архитектуры классов, представляющих данные об адресах и телефонах. Затем нужно написать программный код этих классов, обеспечивающий хранение соответствующей информации и соблюдение бизнес-правил по их формату. Например, в Северной Америке все телефонные номера имеют не более 10 цифр, а почтовые индексы должны записываться по определенным правилам.

Но лишь только вы создали код новых классов, как внезапно сталкиваетесь с тем, что приложение должно работать и с контактной информацией в других форматах, например, в формате, принятом в Нидерландах. В этой связи вам приходится внести изменения в логику классов Address и PhoneNumber, чтобы можно было работать с контактной информацией лиц, находящихся в этой стране.

Но по мере роста ваших связей вам то и дело приходится сталкиваться с новыми форматами представления адресов и телефонов. Чем больше бизнес-правил вы добавляете в базовые классы Address и PhoneNumber, тем труднее становится управлять разросшимся программным кодом. Более того, этот код еще нужно отлаживать — добавляя новые бизнес-правила для очередной страны, вам приходится снова и снова модифицировать и перекомпилировать классы.

Ясно, что в данной ситуации хорошо было бы иметь гибкий механизм добавления подобных классов к уже работающей и отлаженной системе, которые бы использовали общие правила работы с данными об адресах и номерах телефонов и позволяли бы "подгружать" по мере необходимости любое количество вариантов форматов других стран.

Данную проблему позволяет решить шаблон Abstract Factory. В нашем примере с его помощью можно определить универсальную среду генерации объектов, соответствующих общему шаблону для классов Address и PhoneNumber, назвав ее, к примеру, AddressFactory. Во время работы приложения эта среда порождает любое количество конкретных классов для работы с форматами различных стран, и при этом для каждой страны создается своя версия классов Address и PhoneNumber.

Теперь, вместо того, чтобы мучится с добавлением функциональной логики к старым классам, вы расширяете класс Address до класса DutchAddress, а класс PhoneNumber — до класса DutchPhoneNumber. Экземпляры обоих классов создаются с помощью DutchAddressFactory. Такой подход дает гораздо больше свободы для наращивания возможностей приложения без необходимости внесения значительных структурных модификаций в уже работающую систему.

Область применения

Шаблон Abstract Factory рекомендуется использовать в следующих случаях.

- Клиент не должен зависеть от способа получения продукта.
- Приложение должно быть настраиваемым на использование одного из нескольких семейств продукта.
- Объекты должны создаваться в виде определенного набора, чтобы обеспечивалась возможность их сравнения.
- Необходимо создать набор классов и открыть для доступа их параметры и взаимосвязи, не открывая детали их реализации.

Описание

В некоторых случаях приложение должно работать с широкой гаммой ресурсов или в разнородных операционных системах. Вот лишь некоторые, наиболее типичные примеры:

- оконный интерфейс;
- файловая система;
- коммуникации с другими приложениями или системами.

При разработке таких приложений возникает необходимость в создании механизма, обеспечивающего надлежащий уровень их гибкости, чтобы подобные приложения могли работать с разными ресурсами. В то же время необходимо позаботиться и о том, чтобы при добавлении нового ресурса не приходилось перерабатывать исходный код уже работающего приложения.

Пожалуй, самый эффективный способ решения данной проблемы состоит в определении универсального механизма создания ресурсов, представленного шаблоном Abstract Factory. Этот механизм обладает одним или более методов создания объектов, к которым можно обращаться для генерации базовых ресурсов или абстрактных продуктов.

Примером может быть сама технология Java, реализованная на многих платформах, для каждой из которых имеется много различных файловых систем или оконных интерфейсов. Решение, принятое разработчиками технологии Java, состоит в абстракции концепции файлов и оконного интерфейса и сокрытии деталей конкретной реализации. Это позволяет создавать приложения, использующие базовые возможности ресурсов так, как если бы они представляли реальную функциональность.

Во время выполнения приложение создает и использует конкретные экземпляры механизмов (ConcreteFactory) и порожденные ими конкретные продукты (ConcreteProducts). Получаемые классы полностью соответствуют *требованиям* (contract), определенным абстрактными механизмами вида AbstractFactory и AbstractProducts, что обеспечивает возможность прямого использования этих конкретных классов без повторного кодирования или перекомпиляции.

Реализация

Диаграмма классов шаблона Abstract Factory представлена на рис. 1.1.

При реализации шаблона Abstract Factory обычно используются следующие классы.

- AbstractFactory. Абстрактный класс или интерфейс, который определяет механизмы создания абстрактных продуктов.
- `AbstractProduct`. Абстрактный класс или интерфейс, который описывает базовые функции ресурса, который будет использоваться приложением.
- ConcreteFactory. Класс, порожденный от класса абстрактного механизма и реализующий методы получения одного или нескольких конкретных продуктов.
- `ConcreteProduct`. Класс, порожденный от класса абстрактного продукта и реализующий определенный ресурс или операционную среду.

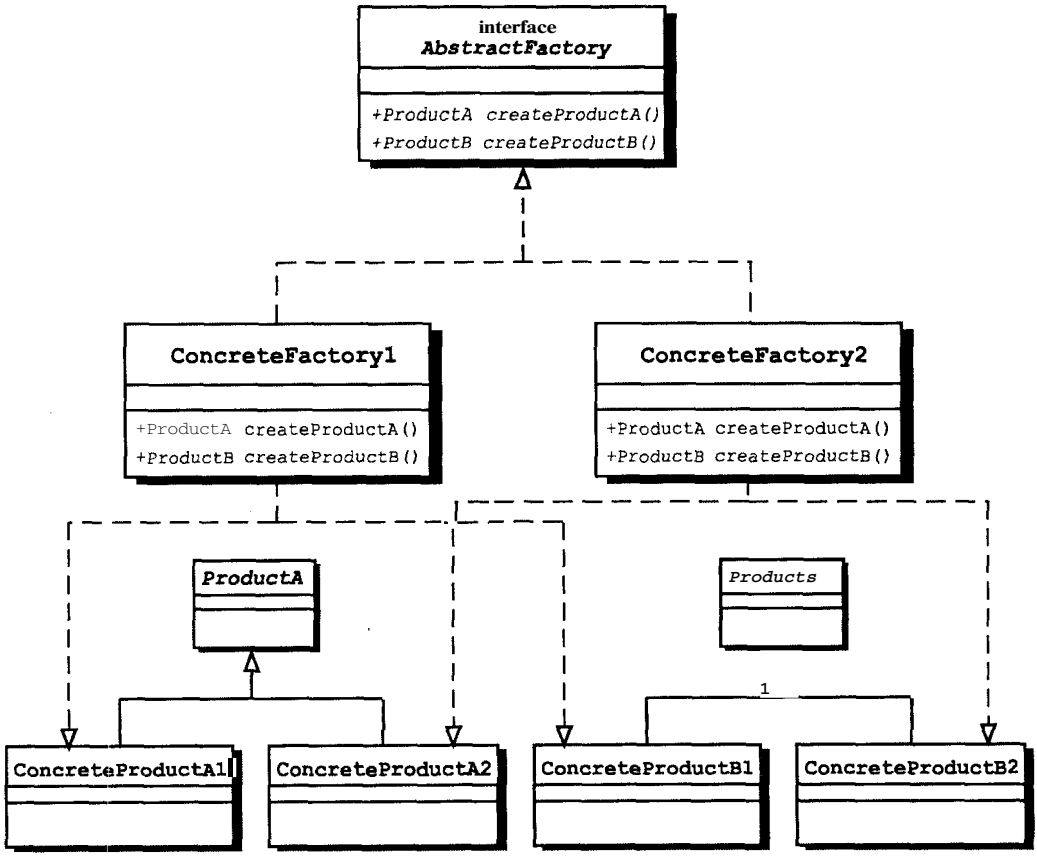


Рис. 1.1. Диаграмма класса Abstract Factory

Достоинства и недостатки

Шаблон Abstract Factory помогает повысить общую гибкость приложения. При этом гибкость должна проявляться, как во время разработки приложения, так и во время его работы. Действительно, проектируя приложение, далеко не всегда можно предсказать его дальнейшую область применения. При использовании шаблона Abstract Factory в этом и нет особой нужды: достаточно определить общий "каркас", а затем сосредоточиться на реализации функциональности отдельных частей независимо от остальных компонентов приложения. Такой подход обязательно проявиться и во время работы приложения — с ним будет гораздо легче интегрировать новые компоненты и ресурсы.

Еще одно преимущество данного шаблона состоит в том, что он упрощает тестирование приложения в целом. Например, можно реализовать классы `TestConcreteFactory` и `TestConcreteProducts`, которые могут эмулировать поведение подключаемого к приложению нового ресурса.

Однако для того чтобы данные преимущества стали ощутимыми на практике, необходимо тщательно продумать, как определить соответствующий базовый интер-

фейс абстрактного продукта. Если свойства абстрактного продукта определены небрежно, это может в некоторых случаях существенно усложнить генерацию конкретных продуктов с заданными свойствами или вообще воспрепятствовать их генерации.

Варианты

Как уже упоминалось выше, AbstractFactory и AbstractProducts могут определяться как в виде интерфейса, так и в виде абстрактного класса, в зависимости от потребностей приложения и личных предпочтений разработчика.

Кроме того, можно предусмотреть различные варианты шаблона Abstract Factory в зависимости от способа его использования. Например, отдельные варианты этого шаблона могут генерировать несколько объектов ConcreteFactory, что позволит приложению одновременно использовать несколько семейств объектов Concrete-Products .

Родственныешаблоны

К родственным можно отнести следующие шаблоны.

- Factory Method (стр. 42). Используется для реализации шаблона Abstract Factory.
- Singleton (стр. 54). Часто применяется при реализации конкретных механизмов вида ConcreteFactory.
- Data Access Object [CJ2EEP]. Шаблон Data Access Object может пользоваться шаблоном Abstract Factory, если возникает необходимость повышения гибкости механизмов работы с базами данных.

Примечание

Обозначение CJ2EEP говорит о том, что этот шаблон в данной книге подробно не рассматривается. Полное название использованного источника информации приведено в Приложении Б.

Пример

Ниже приведен программный код, на примере которого показано, как с использованием шаблона Abstract Factory можно организовать поддержку иностранных форматов адресов и номеров телефонов в PIM-приложении. Сам механизм создается с помощью интерфейса AddressFactory (листинг 1.1).

Листинг 1.1. AddressFactory.java

```
1. public interface AddressFactory{
2.  public Address createAddress() ;
3.  public PhoneNumber createPhoneNumber() ;
4. }
```

Обратите внимание на то, что в интерфейсе AddressFactory определяется два метода, обеспечивающих работу механизма: createAddress и createPhoneNumber. Эти методы генерируют абстрактные продукты Address (листинг 1.2) и PhoneNumber (листинг 1.3), на которые и возлагается задача определения функциональности самих продуктов.

Листинг 1.2. Address.java

```
1. public abstract class Address{
2.   private String street;
3.   private String city;
4.   private String region;
5.   private String postalCode;
6.
7.   public static final String EOL_STRING =
8.      System.getProperty("line.separator");
9.   public static final String SPACE = " ";
10.
11.  public String getstreet(){ return street; }
12.  public String getCity(){ return city; }
13.  public String getPostalCode(){ return postalCode; }
14.  public String getRegion(){ return region; }
15.  public abstract String getCountry();
16.
17.  public String getFullAddress(){
18.     return street + EOL_STRING +
19.       city + SPACE + postalCode + EOL_STRING;
20. }
21.
22.  public void setStreet(String newStreet){ street = newStreet; }
23.  public void setCity(String newCity){ city = newCity; }
24.  public void setRegion(String newRegion){ region = newRegion; }
25.  public void setPostalCode(String newPostalCode){ postalCode =
   newPostalCode; }
26.}
```

Листинг 1.3. PhoneNumber.java

```
1. public abstract class PhoneNumber{
2.   private String phoneNumber;
3.   public abstract String getCountryCode();
4.
5.   public String getPhoneNumber(){ return phoneNumber; }
6.
7.   public void setPhoneNumber(String newNumber){
8.      try{
9.        Long.parseLong(newNumber) ;
10.       phoneNumber = newNumber;
11.     }
12.     catch (NumberFormatException exc){
13.     }
14. }
15.}
```

В рассматриваемом примере Address и PhoneNumber — это абстрактные классы, но в тех ситуациях, когда нет необходимости помещать в них код, предназначенный для работы во всех конкретных продуктах, их можно определить в виде интерфейсов.

Для того чтобы обеспечить в разрабатываемой системе какую-то конкретную функциональность, необходимо создать классы конкретного механизма и конкретного продукта. В нашем примере мы определим класс USAddressFactory (листинг 1.4), который реализует интерфейс AddressFactory, а также подклассы USAddress (листинг 1.5) и USPhoneNumber (листинг 1.6) классов Address и PhoneNumber, соответственно.

Листинг 1.4. USAddressFactory.java

```
1. public class USAddressFactory implements AddressFactory{
2.   public Address createAddress(){
3.     return new USAddress ();
4.   }
5.
6.   public PhoneNumber createPhoneNumber(){
7.     return new USPhoneNumber();
8.   }
9. }
```

Листинг 1.5. USAddress.java

```
1. public class USAddress extends Address{
2.   private static final String COUNTRY = "UNITED STATES";
3.   private static final String COMMA = ",";
4.
5.   public String getCountry(){ return COUNTRY; }
6.
7.   public String getFullAddress(){
8.     return getStreet()  + EOL_STRING +
9.       getCity() + COMMA + SPACE + getRegion() +
10.      SPACE + getPostalCode() + EOL_STRING +
11.      COUNTRY + EOL_STRING;
12. }
13.}
```

Листинг 1.6. USPhoneNumber.java

```
1. public class USPhoneNumber extends PhoneNumber)
2.   private static final String COUNTRY_CODE = "01";
3.   private static final int Number_LENGTH = 10;
4.
5.   public String getCountryCode(){ return COUNTRY_CODE; }
6.   public void setPhoneNumber(String newNumber){
7.     if (newNumber.length() == NUMBER_LENGTH){
8.       super.setPhoneNumber) ;
9.     }
10. }
11.}
```

Полученный набор классов AddressFactory, Address и PhoneNumber облегчает процесс расширения возможностей системы для поддержки форматов адресов и номеров телефонов, принятых в других странах. Теперь для того, чтобы добавить функции поддержки форматов какой-либо дополнительной страны, достаточно определить конкретный класс механизма и соответствующий ему конкретный класс продукта. Например, в листингах 1.7-1.9 приведены классы, которые можно использовать для добавления поддержки форматов, принятых во Франции.

Листинг 1.7. FrenchAddressFactory.java

```
1. public class FrenchAddressFactory implements AddressFactory{
2.   public Address createAddress(){
3.      return new FrenchAddress();
4.   (
5.
6.   public PhoneNumber createPhoneNumber(){
7.      return new FrenchPhoneNumber();
8.   }
9. }
```

Листинг 1.8. FrenchAddress.java

```
1. public class FrenchAddress extends Address{
2.   private static final String COUNTRY = "FRANCE";
3.
4.   public String getCountry(){ return COUNTRY; )
5.
6.   public String getFullAddress(){
7.      return getStreet() + EOL_STRING +
8.       getPostalCode()  + SPACE + getCity() +
9.       EOL_STRING + COUNTRY + EOL_STRING;
10. }
11.}
```

Листинг 1.9. FrenchPhoneNumber.java

```
1. public class FrenchPhoneNumber extends PhoneNumber{
2.   private static final String COUNTRY_CODE = "33";
3.   private static final int NUMBER_LENGIH = 9;
4.
5.   public String getCountryCode(){ return COUNTRY_CODE; }
6.
7.   public void setPhoneNumber(String newNumber){
8.     if (newNumber.length() == NUMBER_LENGTH{
9.        super.setPhoneNumber(newNumber);
10.    }
11. }
12.}
```

Builder

Свойства шаблона

Тип: производящий шаблон
Уровень: компонент

Назначение

Упрощает создание сложных объектов путем определения класса, предназначенного для построения экземпляров другого класса. Шаблон Builder генерирует только одну сущность. Хотя эта сущность, в свою очередь, может содержать более одного класса, но один из полученных классов всегда является главным.

Представление

В состав большинства PIM-приложений входит подсистема, обеспечивающая пользователю возможность работы с календарем. Для реализации такой подсистемы в разрабатываемом приложении можно определить специальный класс, например Appointment, который будет обеспечивать хранение информации об определенном событии, а также работу с разными формами ее представления, например:

- дата начала и завершения события;
- описание события;
- место проведения события;
- список приглашенных лиц.

Естественно, все подобные сведения должны вводиться пользователем при планировании события, поэтому необходимо определить конструктор, который обеспечил бы установку состояния нового объекта класса Appointment.

Таким образом, необходимо дать ответ на вопрос: "Какая именно информация обязательно должна сохраняться приложением при планировании события?" Однозначно на него ответить нельзя, так как у разных типов событий разные наборы сведений, относящихся к обязательным. Например, при планировании определенных событий самое главное — это список приглашенных (например, ежемесячное собрание закрытого фан-клуба какой-нибудь кинозвезды). Для других событий важнее всего даты их начала и окончания (например, конференция JavaOne). Для третьих же, таких как планируемое пользователем посещение в отпуске вернисажа модного художника, достаточно лишь одной даты начала события. Поэтому задача создания объекта Appointment уже не кажется тривиальной.

Хотя существует два подхода к управлению созданием объектов, ни один из них не имеет каких-либо особых преимуществ. Можно либо создавать конструкторы по отдельности для каждого типа событий, либо разработать один универсальный и всеобъемлющий конструктор, насыщенный до предела функциональной логикой. У каждого из этих подходов, как уже было сказано, имеются свои недостатки. В случае ис-

пользования множества конструкторов значительно усложняется логика их вызовов, а в случае использования одного развитого конструктора резко повышается сложность программного кода и затрудняется его отладка. Что еще хуже, как в том, так и в другом случае при последующем развитии приложения разработчик, создавая новые классы на основе класса Appointment, скорее всего столкнется с теми или иными проблемами.

Если же делегировать ответственность за создание объектов класса Appointment специальному классу, например `AppointmentBuilder`, это позволит значительно упростить программный код самого класса Appointment. Класс AppointmentBuilder обладает методами, обеспечивающими создание различных составляющих класса Appointment. Программисту лишь остается вызвать в нужный момент тот или иной метод класса AppointmentBuilder, в соответствии с выбранным пользователем типом планируемого события. Кроме того, класс AppointmentBuilder может проверить корректность информации, которая передается создаваемому объекту класса Appointment, тем самым облегчая задачу обеспечения бизнес-правил. Для того чтобы создать подкласс класса Appointment, можно либо создать новый класс Builder, либо создать подкласс на основе существующего класса. В обоих случаях решить подобную задачу гораздо легче, чем управлять инициализацией объектов с помощью множества конструкторов.

Область применения

Шаблон Builder рекомендуется использовать в следующих случаях.

- Класс имеет сложную внутреннюю структуру (особенно это касается тех классов, которые работают с переменными наборами связанных с ними объектов).
- Атрибуты класса зависят один от другого. Одно из типичных применений шаблона Builder — это его применение в тех случаях, когда имеет место поэтапное создание сложного объекта. Если атрибуты продукта зависят один от другого, его невозможно получить другим способом. Например, предположим, что мы создаем такой объект, как заказ. В этом случае, прежде чем переходить к "созданию" метода доставки, необходимо убедиться в том, что установлено значение, определяющее штат США, так как оно влияет на значение налога с продаж, которое применяется ко всему заказу.
- Класс использует другие объекты системы, получить которые во время создания затруднительно или неудобно.

Описание

Так как данный шаблон предназначен для построения сложных объектов, причем необязательно из одного источника, он называется Builder. Усложнение способов создания объекта ведет к возрастанию сложности управления созданием объекта с помощью конструкторов. Это особенно относится к тем объектам, которые не могут полагаться лишь на ресурсы, которые находятся под их контролем.

К такой категории часто можно отнести объекты бизнес-модели. Как правило, таким объектам требуются для инициализации данные, находящиеся в базе данных, а

также для работы им может понадобиться взаимодействовать с другими объектами, образуя, таким образом, бизнес-модель. В качестве другого примера можно привести комплексные объекты некой системы, такие как объекты, представляющие собой элементы рисунка, которые обрабатываются программой визуального редактирования. Такие объекты могут требовать установления связей с произвольным количеством других объектов, причем сразу же в момент создания.

В подобных случаях имеет смысл определить отдельный класс (Builder), на который возлагается ответственность за создание прикладных объектов. Этот класс координирует сборку объектов-продуктов: создание ресурсов, сохранение промежуточных результатов и обеспечение функциональной структуры создаваемого объекта. Кроме того, Builder может получать системные ресурсы, необходимые для создания объекта-продукта.

Реализация

Диаграмма классов шаблона Builder представлена на рис. 1.2.

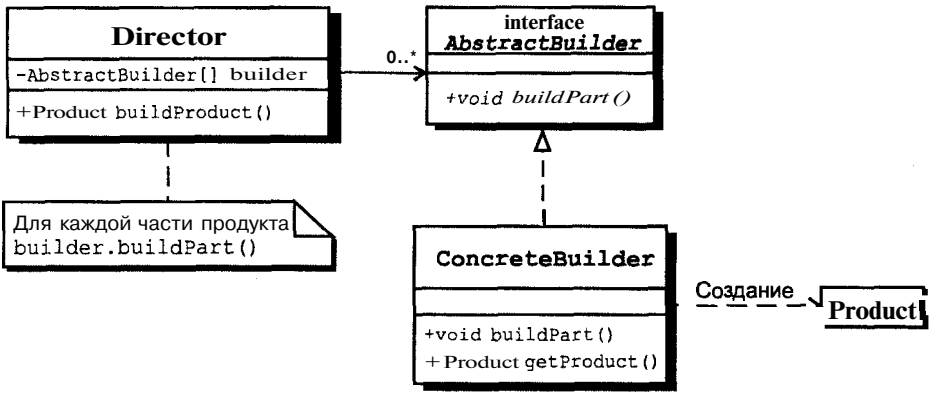


Рис. *1.2.* Диаграмма класса Builder

При реализации шаблона Builder обычно используются следующие классы.

- Director. Содержит ссылку на экземпляр класса AbstractBuilder. По этой ссылке класс Director вызывает методы создания конкретного генератора классов, которые создают различные части продукта, что в конечном итоге приводит к созданию нового объекта-продукта.
- AbstractBuilder. Интерфейс, который определяет методы, предназначенные для создания отдельных частей продукта.
- ConcreteBuilder. Класс, реализующий интерфейс AbstractBuilder. Класс ConcreteBuilder содержит реализацию всех методов, необходимых для создания объекта Product. При реализации методов должна обеспечиваться работа алгоритмов, на которые возлагается задача обработки полученной от класса Director информации и создание соответствующих частей объекта Product. Кроме того, у класса ConcreteBuilder обязательно должен быть

метод `getProduct` либо метод создания нового объекта, который возвращал бы ссылку на новый экземпляр класса `Product`.

- `Product`. Полученный объект. Продукт можно определять и как интерфейс (что предпочтительнее), и как класс.

Достоинства и недостатки

Шаблон Builder облегчает управление процессом создания сложных объектов. Это свойство шаблона проявляется в одной из двух форм.

- Для объектов, которые требуют поэтапного создания (т.е. соблюдения определенной последовательности операций, в результате которых объект становится полноценным), шаблон Builder играет роль объекта более высокого уровня, который управляет всем ходом процесса. Он может координировать создание всех ресурсов и проверять их корректность, а также при необходимости обеспечивать стратегию восстановления в случае возникновения ошибок.
- Для объектов, которым во время создания необходим доступ к уже существующим ресурсам системы, например, когда необходимо подключение к базе данных или доступ к существующим объектам бизнес-модели, шаблон Builder играет роль центрального пульта управления такими ресурсами. Кроме того, в подобных случаях шаблон Builder также является средством централизованного управления созданием генерируемых подобными объектами продуктов, предназначенных для получения к ним доступа со стороны других объектов системы. Как и в случае с другими производящими шаблонами, это упрощает работу клиентов программной системы, так как им нужно для генерации нового ресурса получить доступ лишь к одному объекту Builder.

Основной недостаток этого шаблона состоит в наличии жесткой связи между самим генератором Builder, его продуктом и другими производными объектами, наделенными полномочиями генератора, используемыми при создании объектов. Это приводит к тому, что изменения, вносимые в продукт, генерируемый с помощью шаблона Builder, часто требуют внесения изменений как в сам генератор Builder, так и в наделенные его полномочиями объекты.

Варианты

Теоретически нет никаких препятствий для того, чтобы реализовать минимальный шаблон Builder на основе лишь одного класса `Builder`с методом создания объекта и его продукта. Однако на практике разработчики, стремясь обеспечить повышенный уровень гибкости шаблона, расширяют базовый шаблон с помощью одного или нескольких из следующих приемов.

- Создание абстрактного генератора Builder. Определив абстрактный класс или интерфейс, в котором метод создания объекта просто объявляется, мы получаем более универсальную систему, на основе которой можно реализовывать различные виды генераторов.

- Определение у генератора Builder нескольких методов создания объекта. В некоторых случаях можно прибегнуть к определению нескольких методов (это достигается путем использования перегрузки методов генератора, предназначенных для создания объектов), что позволяет получить несколько разных способов инициализации конструируемого ресурса.
- Разработка объектов, наделенных полномочиями генератора. В данном варианте объект Director содержит общий метод создания объектов Product, вызов которого сводится к серии вызовов более специализированных методов создания конкретного генератора Builder. Таким образом, при использовании данного подхода объект Director играет роль диспетчера, управляя процессом создания объектов генератором Builder.

Родственные шаблоны

К родственным можно отнести шаблон Composite (стр. 171). На практике шаблон Builder часто используется для генерации именно объектов Composite, так как последние имеют очень сложную структуру.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Builder" на стр. 350 Приложения А.

В приведенном ниже примере показано, как можно использовать шаблон Builder для создания объекта, представляющего в нашем PIM-приложении некое запланированное пользователем событие. Использованные в примере классы имеют следующее назначение.

- `AppointmentBuilder`, `MeetingBuilder` — классы генераторов.
- Scheduler — класс диспетчера.
- Appointment — продукт.
- Address, Contact — вспомогательные классы, используемые для хранения информации, необходимой классу Appointment.
- `InformationRequiredException` — класс исключительной ситуации Exception, который генерируется в случае нехватки необходимых данных.

Основа шаблона — это класс AppointmentBuilder (листинг 1.10), который управляет созданием комплексного продукта, представленного в данном примере объектом Appointment (листинг 1.11). Класс AppointmentBuilder использует серию методов создания отдельных элементов события (`buildAppointment`, buildLocation, buildDates и buildAttendees), в результате вызова которых формируются объект Appointment с соответствующими данными.

Листинг 1.10. AppointmentBuilder.java

```
1. import java.util.Date;
2. import java.util.ArrayList;
3.
4. public class AppointmentBuilder{
5.
6.   public static final int START_DATE_REQUIRED = 1;
7.   public static final int END_DATE_REQUIRED = 2;
8.   public static final int DESCRIPTION_REQUIRED = 4;
9.   public static final int ATTENDEE_REQUIRED = 8;
10.  public static final int LOCATION_REQUIRED = 16;
11.
12.  protected Appointment appointment;
13.
14.  protected int requiredElements;
15.
16.  public void buildAppointment(){
17.    appointment = new Appointment ();
18. }
19.
20.  public void buildDates(Date startDate, Date endDate){
21.     Date currentDate = new Date();
22.     if ((startDate != null) 44 (startDate.after(currentDate))){
23.       appointment.setStartDate(startDate) ;
24.     }
25.     if ((endDate != null) && (endDate.after(startDate))){
26.       appointment.setEndDate(endDate) ;
27.     }
28. }
29.
30.  public void buildDescription(String newDescription){
31.     appointment.setDescription(newDescription);
32. )
33.
34.  public void buildAttendees(ArrayList attendees){
35.     if ((attendees != null) && (!attendees.isEmpty())){
36.       appointment.setAttendees(attendees) ;
37.     }
38. }
39.
40.  public void buildLocation(Location newLocation)(
41.     if (newLocation != null){
42.       appointment.setLocation(newLocation) ;
43.     }
44. }
45.
46.  public Appointment getAppointment() InformationRequiredException(
47.    requiredElements = 0;
48.
49.     if (appointment.getStartDate() == null){
50.       requiredElements += START_DATE_REQUIRED;
51.     }
52.
53.     if (appointment.getLocation() == null){
54.       requiredElements += LOCATION_REQUIRED;
55.     }
56.
57.     if (appointment.getAttendees().isEmpty()){
58.       requiredElements += ATTENDEE_REQUIRED;
59.     }
60.
```

```
61.     if (requiredElements > 0)(
62.       throw new InformationRequiredException(requiredElements);
63.     }
64.     return appointment;
65. }
66.
67. public int getRequiredElements(){ return requiredElements; }
68.}
```

Листинг 1.11. Appointment.java

```
1. import java.util.ArrayList;
2. import java.util.Date;
3. public class Appointment{
4.   private Date startDate;
5.   private Date endDate;
6.   private String description;
7.   private ArrayList attendees = new ArrayList ();
8.   private Location location;
9.   public static final String EOL_STRING =
10.    System.getProperty("line.separator") ;
11.
12.  public Date getStartDate(){ return startDate; )
13.  public Date getEndDate(){ return endDate; }
14.  public String getDescription(){ return description; }
15.  public ArrayList getAttendees(){ return attendees; }
16.  public Location getLocation(){ return Location; )
17.
18.  public void setDescription(String new Description){ description =
   newDescription; )
19.  public void setLocation(Location newLocation){ location = newLocation; }
20.  public void setStartDate (Date newStartDate){ startDate = newStartDate; }
21.  public void setEndDate(Date newEndDate){ endDate = newEndDate; }
22.  public void setAttendees(ArrayList newAttendees){
23.     if (newAttendees != null){
24.       attendees = newAttendees;
25.     }
26. )
27.
28.  public void addAttendee(Contact attendee){
29.     if (!attendees.contains(attendee)){
30.       attendees.add(attendee) ;
31.     }
32.  }
33.
34.  public void removeAttendee(Contact attendee){
35.     attendees.remove(attendee);
36.  }
37.
38.  public String toString(){
39.     return " Description: " + description + EOL_STRING +
40.       " Start Date: " + startDate + EOL_STRING +
41.       " End Date: " + endDate + EOL_STRING +
42.       " Location: " + Location + EOL_STRING +
43.       " Attendees: " + attendees;
44.  }
45.}
```

Класс Scheduler (листинг 1.12) вызывает методы класса AppointmentBuilder, управляя ходом процесса с помощью метода createAppointment.

Листинг 1.12. Scheduler.java

```
1. import java.util.Date;
2. import java.util.ArrayList;
3. public class Scheduler{
4.  public Appointment createAppointment(AppointmentBuilder builder,
5.    Date startDate, Date endDate, String description,
6.    Location location, ArrayList attendees) throws
   InformationRequiredException{
7.       if (builder == null){
8.       builder = new AppointmentBuilder();
9.        }
10.     builder.buildAppointment();
11.    builder.buildDates(startDate, endDate);
12.     builder.buildDescription(description);
13.    builder.buildAttendees(attendees) ;
14.     builder.buildLocation(Location);
15.     return builder.getAppointment();
16. }
17.}
```

Таким образом, классы решают следующие задачи.

- Scheduler. Вызывает нужные методы класса AppointmentBuilder, необходимые для создания объекта Appointment; возвращает ссылку на созданный объект Appointment.
- AppointmentBuilder. Содержит методы создания элементов объекта и применения бизнес-правил; выполняет собственно генерацию объекта Appointment.
- Appointment. Содержит информацию о событии.

Приведенный в листинге 1.13 класс MeetingBuilder позволяет увидеть на практике одно из преимуществ шаблона Builder, состоящее в том, что для добавления дополнительных правил к классу Appointment достаточно расширить уже имеющийся генератор. В данном примере класс MeetingBuilder накладывает на планируемое событие дополнительные ограничения, а именно: при планировании такого события, как деловая встреча, должны быть обязательно указаны данные о времени начала и конца этого события.

Листинг 1.13. MeetingBuilder.java

```
1. import java.util.Date;
2. import java.util.Vector;
3.
4. public class MeetingBuilder extends AppointmentBuilder{
5.  public Appointment getAppointment() throws InformationRequiredException{
6.     try{
7.       super.getAppointment();
8.      )
9.     finally{
10.      if (appointment.getEndDate() == null){
11.           requiredElements += END_DATE_REQUIRED;
```

```
12.       }
13.
14.       if (requiredElements > 0){
15.           throw new InformationRequiredException(requiredElements) ;
16.       }
17.     }
18.     return appointment;
19.   }
20.}
```

Factory Method

Также известен как Virtual Constructor

Свойства шаблона

Тип: производящий шаблон
Уровень: класс

Назначение

Определяет стандартный метод создания объекта, не связанный с вызовом конструктора, оставляя решение о том, какой именно объект создавать, за подклассами.

Представление

Продолжим рассмотрение нашего примера с **PIM-приложением**. Очевидно, что такое приложение должно манипулировать самой разной информацией, необходимой в повседневной работе: адресами, сведениями о запланированных событиях, датами, записками и т.п. Причем характерной особенностью этой информации является то, что она нестатична. Например, если ваш знакомый переехал, вам захочется заменить его старый адрес, хранимый PIM-приложением, на новый.

Естественно, **PIM-приложение** должно обеспечивать возможность внесения изменений в любое поле. Таким образом, на его разработчика возлагается обязанность предоставить пользователю возможность редактирования (и тем самым позаботится о создании соответствующего пользовательского интерфейса), а также проверки корректности введенного значения для каждого поля. Однако есть и обратная сторона медали — **PIM-приложению** придется обладать средствами для работы со всеми типами событий и задач, которые только может запланировать пользователь. Так как каждый элемент, хранимый персональным планировщиком, в общем случае может иметь какие-то свои уникальные поля, необходимо позаботиться о том, чтобы пользователю каждый раз было предоставлено отдельное диалоговое окно для работы именно с этими полями. Кроме того, это затрудняет добавление информации о заданиях нового типа, так как вместе с ними придется добавлять средства редактирования в само PIM-приложение, которые умеют работать с элементами нового типа. Что еще хуже, любое изменение в каком-то типе задания, такое как добавление нового поля к информации о запланированном событии, повлечет за собой неизбежное обновление

PIM-приложения, чтобы оно могло работать с этим новым полем. В конце концов это приведет к непомерному разрастанию исходного кода PIM-приложения и превращению его в трудно управляемый проект.

Решение состоит в том, чтобы возложить задачу обеспечения модификации и добавления информации на объекты PIM-приложения, представляющие те или иные задания, например запланированные пользователем события. В этом случае планировщик)' знать нужно лишь, как вызвать редактор, воспользовавшись методом `getEditor`, которым должен обладать любой элемент, допускающий редактирование. Метод возвращает объект, реализующий интерфейс `ItemEditor`, а планировщик в свою очередь использует полученный объект для вызова `JComponent` в качестве редактора с пользовательским интерфейсом. Другими словами, пользователи могут нужным им образом модифицировать информацию, относящуюся к выбранному элементу, причем редактор не только обеспечит внесение изменений, но и проверит корректность внесенных данных.

Вся информация о том, как редактировать тот или иной элемент, содержится в соответствующем данному элементу редакторе. Кроме того, редактор отвечает и за свое графическое представление. Такой подход позволяет добавлять новые типы данных, не сталкиваясь с необходимостью внесения изменений в PIM-приложение.

Область применения

Шаблон Factory Method рекомендуется использовать в следующих случаях.

- Необходимо создать систему, обладающую свойством расширяемости. Это означает, что разработчику придется сделать выбор в пользу гибкости, оставляя некоторые решения, например, какой именно объект создавать, "на потом".
- По некоторым соображениям предпочтительнее, чтобы решение о том, какой объект создавать, принималось на уровне подкласса, а не суперкласса.
- В силу тех или иных обстоятельств, известно, когда нужно создать объект, но неизвестно, какой именно.
- Необходимо иметь в распоряжении несколько перекрытых конструкторов с одинаковыми списками параметров, что недопустимо в языке Java. В этом случае можно прибегнуть к шаблону Factory Method, используя разные имена классов.

Описание

Название шаблона Factory Method объясняется тем, что он "фабрикует" объекты, когда в них возникает необходимость.

Приступая к разработке приложения, далеко не всегда можно заранее решить, какие именно компоненты вам понадобятся. Обычно у разработчика есть лишь общее видение того, что должны делать компоненты, но реализация функциональности компонентов с уточнением их возможностей выполняется позже, в ходе работы над проектом.

Конечно, частично данную проблему решает использование интерфейсов, описывающих подобные компоненты. Однако применение интерфейсов в свою очередь затрудняет работу программиста, так как из интерфейса невозможно создать объект. Для того чтобы получить объект, нужно реализовать класс. Поэтому, вместо того,

чтобы разрабатывать реализацию некоего класса, специфичного для конкретного приложения, можно просто вычленить из класса функциональность конструктора и реализовать ее в виде специального метода, "фабрикующего" объекты приложения.

Таким образом можно получить некий класс, например ConcreteCreator, отвечающий за создание определенных объектов. Этот класс предназначается для создания экземпляров реализации (ConcreteProduct) определенного интерфейса (Product).

Реализация

Диаграмма классов шаблона Factory Method представлена на рис. 1.3.

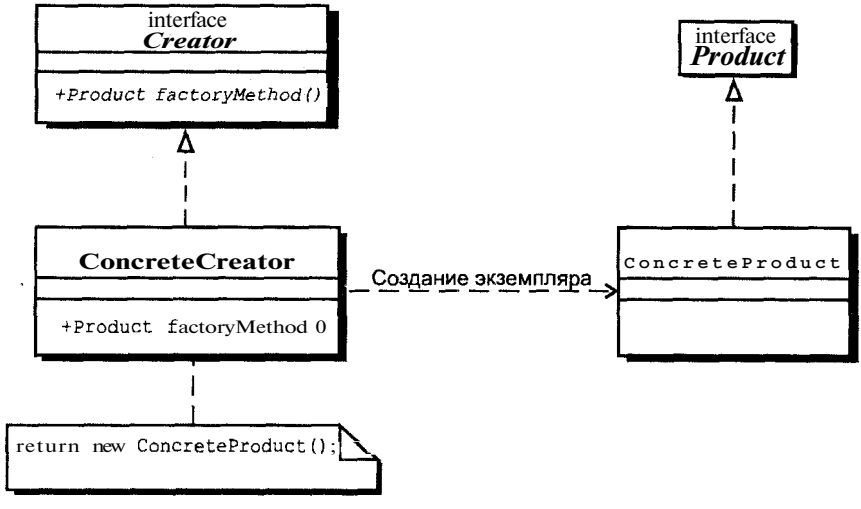


Рис. 1.3. Диаграмма классов шаблона Factory Method

Для реализации шаблона Factory Method необходимы следующие классы.

- Product. Интерфейс "фабрикуемых" объектов.
- ConcreteProduct. Реализация класса Product. Создание объектов этого класса возлагается на класс ConcreteCreator.
- Creator. Интерфейс, который определяет собственно метод, "фабрикующий" объекты (factoryMethod).
- ConcreteCreator. Класс, расширяющий интерфейс Creator и содержащий реализацию метода factoryMethod. Этот метод может возвращать любой объект, который реализует интерфейс Product.

Достоинства и недостатки

Основным достоинством такого решения является обеспечиваемая им универсальность PIM-приложения. В самом деле, приложению достаточно знать лишь то, как вызывается редактор любого элемента, а сведения о том, как нужно редактировать данные этого элемента, содержатся в самом редакторе. Кроме того, на редактор можно возложить и задачу создания графического пользовательского интерфейса, с помощью которого обеспечивается собственно редактирование. Все это позволяет сделать PIM-приложение модульным, упростить добавление к нему информации о новых типах данных и избавиться при этом от необходимости внесения изменений в исходный текст приложения при каждом таком добавлении.

Шаблон Factory Method широко используется во многих интерфейсах системы JDBC (Java Database Connectivity — связь с базами данных из Java). Это позволяет заменять любой JDBC-драйвер любым другим JDBC-драйвером (лишь бы он был корректно написан), не внося изменений в само приложение. (Более подробные сведения о применении шаблонов в системе JDBC приведены в разделе JDBC на стр. 316.)

Варианты

Данный шаблон может выполняться в следующих вариантах.

- Creator может содержать стандартную реализацию метода `factoryMethod`. В этом случае `Creator` уже не должен быть абстрактным классом или интерфейсом, а может выполняться в виде обычного класса. Достоинством этого варианта является то, что разработчик избавляется от необходимости создавать подкласс класса `Creator`.
- Product может выполняться в виде абстрактного класса, что позволяет разместить в нем реализацию других методов.
- Методу, "фабрикующему" объекты, можно передавать параметр. Это позволяет генерировать продукты разных типов в зависимости от полученного параметра, что приводит к снижению количества необходимых методов вида factoryMethod.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Abstract Factory (стр. 26). Может использовать один или несколько методов типа `factoryMethod`.
- Prototype (стр. 48). Позволяет избавиться от необходимости создания подкласса класса Creator.
- Template Method (стр. 146). Шаблонные методы обычно вызывают методы, "фабрикующие" объекты.
- Data Access Object [CJ2EEP]. Шаблон Data Access Object использует шаблон Factory Method, чтобы обеспечить возможность создания конкретных экземпляров Data Access Object без привязки к особенностям нижележащей базы данных.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`*, приведен в разделе "Factory Method" на стр. 358 Приложения* ***А.***

В приведенном ниже примере показано, как для обеспечения возможности редактирования элемента **PIM-приложения** используется шаблон Factory Method. Такая возможность, очевидно, весьма пригодится пользователю PIM-приложения, которое по определению должно управлять самой разной информацией. Для повышения гибкости системы в рассматриваемом приложении используются интерфейсы.

В интерфейсе Editable (листинг 1.14) объявлен метод `getEditor`, возвращающий интерфейс `ItemEditor`. Достоинством использования интерфейса Editable является то, что с его помощью любой элемент PIM-приложения может предоставить пользователю редактор для изменения своих данных. Это обеспечивается путем генерации объекта, которому известно, какие данные прикладного объекта можно изменять и какие значения являются допустимыми. Единственная задача, которая возлагается на подсистему пользовательского интерфейса, — это использование интерфейса Editable для получения редактора.

Листинг 1.14. Editable.java

```
1. public interface Editable {
2.   public ItemEditor getEditor();
3. }
```

В интерфейсе ItemEditor (листинг 1.15) в свою очередь объявлены два метода: `getGUI` и commitChanges. Метод `getGUI` — это еще один метод, "фабрикующий" объекты. В данном случае он генерирует объекты класса `JComponent`, которые представляют собой графический пользовательский интерфейс Swing и обеспечивают редактирование текущего элемента. Это позволяет получить очень гибкую систему: при добавлении элемента нового типа графический пользовательский интерфейс может оставаться неизменным, так как он использует лишь интерфейсы Editable и ItemEditor.

Объект JComponent, возвращаемый методом getGUI, может обладать всеми средствами, необходимыми для редактирования выбранного элемента PIM-приложения. Подсистеме пользовательского интерфейса остается лишь отобразить полученный объект JComponent, чтобы обеспечить пользователю возможность применения функциональности JComponent для редактирования элемента. Попутно заметим, что так как далеко не каждое приложение работает в графическом режиме, было бы неплохо предусмотреть и наличие альтернативного метода, такого как `getUI`, возвращающего объект класса `Object` или другого класса, обеспечивающего пользователю возможность работы с данными выбранного элемента без графического интерфейса.

Второй метод, commitChanges, позволяет пользовательскому интерфейсу сообщить редактору о том, что пользователь решил завершить внесение изменений.

Листинг 1.15. ItemEditor.java

```
1. import javax.swing.JComponent;
2. public interface ItemEditor {
3.   public JComponent getGUIO;
4.   public void commitChanges();
5. }
```

В листинге 1.16 показан пример возможной реализации одного из элементов PIM-приложения, Contact. Класс Contact имеет два атрибута: имя контактного лица и указание на то, кем оно доводится пользователю. На примере этих атрибутов показано, как можно вводить информацию, описывающую элемент PIM-приложения.

Листинг 1.16. Contact.java

```
1. import java.awt.GridLayout;
2. import java.io.Serializable;
3. import javax.swing.JComponent;
4. import javax.swing.JLabel;
5. import javax.swing.JPanel;
6. import javax.swing.JTextField;
7.
8. public class Contact implements Editable, Serializable {
9.   private String name;
10.  private String relationship;
11.
12. public ItemEditor getEditor () {
13.    return new ContactEditor();
14. }
15.
16.  private class ContactEditor implements ItemEditor, Serializable {
17.    private transient JPanel panel;
18.    private transient JTextField nameField;
19.    private transient JTextField relationField;
20.
21.    public JComponent getGUI() {
22.      if (panel ==null)  (
23.          panel = new JPanel ();
24.           nameField = new JTextField(name);
25.           relationField = new JTextField(relationship);
26.           panel.setLayout(new GridLayout(2,2));
27.           panel.add(new JLabel("Name:"));
28.          panel.add(nameField) ;
29.           panel.add(new JLabel("Relationship:"));
30.          panel.add(relationField) ;
31.      } else {
32.          nameField.setText(name) ;
33 .          relationField.setText(relationship);
34.      }
35.      return panel;
36.    }
37.
38.    public void commitChanges()  {
39.      if (panel != null)  {
40.            name = nameField.getText();
41.            relationship = relationField.getText();
42.      }
43.    }
44.
```

```
45.    public String toString(){
46.      return "\nContact:\n" +
47.           "    Name: " + name + "\n" +
48.           "    Relationship: " + relationship;
49.    }
50. }
51.}
```

Класс Contact реализует интерфейс Editable и предоставляет собственный редактор. Этот редактор может работать лишь с объектами класса Contact, так как он предназначен для изменения определенных атрибутов этого класса. В этой связи он реализован в виде вложенного класса. Такое решение позволяет ему получать доступ к атрибутам внешнего класса. В случае реализации редактора в виде отдельного (т.е. не вложенного) класса пришлось бы обеспечить класс Contact дополнительными методами, с помощью которых редактор мог бы получать доступ к атрибутам и изменять их значения. Понятно, что такое решение некорректно с точки зрения ограничения доступа к внутренним данным класса Contact.

Обратите внимание на то, что редактор сам по себе не является компонентом Swing, а выполнен в виде объекта, который обеспечивает механизм получения такого компонента. Преимущество такого подхода состоит в том, что разработчик может применять для этого объекта сериализацию и направлять его в поток. Для этого необходимо объявить все Swing-атрибуты класса ContactEditor транзитными (transient), и они будут создаваться в тех случаях, когда в этом возникнет необходимость.

Prototype

Свойства шаблона

Тип: производящий шаблон
Уровень: отдельный класс

Назначение

Облегчает динамическое создание путем определения классов, объекты которых могут создавать собственные дубликаты.

Представление

Проектируя функциональность PIM-приложения, нетрудно предположить, что пользователю может понадобиться скопировать адрес контактного лица в новую запись. Чтобы избавить его от ручного копирования, можно решить эту задачу в следующей последовательности:

1. создать новый объект класса Address;
2. скопировать нужные значения из соответствующих полей существующего объекта класса Address.

Данный подход позволяет получить ожидаемый результат, но имеет тот недостаток, что нарушается объектно-ориентированный принцип инкапсуляции. Если же мы будем соблюдать этот принцип, нам придется добавить в класс Address методы, обеспечивающие копирование информации класса с помощью вызовов этих методов извне. В таком случае мы столкнемся с новой проблемой — постоянным повышением сложности сопровождения исходного кода класса Address. Кроме того, это затруднит повторное использование класса Address в будущих проектах.

Но даже если и помещать копирующий код в класс `Address`, почему бы не обеспечить с его помощью "копирование" всего класса? Это позволит обойтись всего лишь одним дополнительным методом копирования, возвращающего дубликат объекта Address со значением данных, аналогичным соответствующим данным оригинала, который, таким образом, выступает прототипом. Вызов всего лишь одного специального метода объекта Address позволяет решить проблему гораздо изящнее и в строгом соответствии с объектно-ориентированными подходами к программированию.

Область применения

Шаблон Prototype рекомендуется использовать в тех случаях, когда необходимо создать объект, являющийся точной копией какого-либо объекта.

Описание

Название шаблона Prototype говорит само за себя: как и любой другой прототип, он относится к объекту, который используется в качестве образца для создания нового экземпляра с такими же значениями полей. Обеспечение возможности "создания по состоянию" позволяет программам выполнять операции подобно управляемому пользователем копированию, а также инициировать объекты, начальное состояние которых было настроено в системе заранее. Такой подход к инициализации во многих случаях, когда нужно получить объект с определенным набором значений, оказывается весьма удобным.

Классические примеры использования этого шаблона можно встретить в графических и текстовых редакторах, в которых функции копирования и вставки в значительной степени повышают производительность труда пользователя. Некоторые бизнес-пакеты также используют данный подход, позволяя создавать новые модели на основе уже существующих. Полученные копии, как правило, модифицируются пользователем для получения нужного ему нового состояния только что созданных объектов.

Реализация

Диаграмма классов шаблона Prototype представлена на рис. 1.4.

При реализации шаблона Prototype используется класс `Prototype`. В этом классе определяется метод копирования, который возвращает экземпляр того же класса с теми же значениями, что и оригинальный экземпляр `Prototype`. Новый экземпляр может быть как частичной (shallow), так и полной (deep) копией оригинального объекта (подробнее см. следующий подраздел "Достоинства и недостатки").

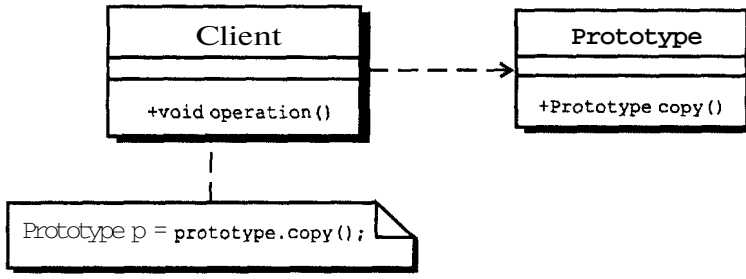


Рис. 1.4. Диаграмма класса Prototype

Достоинства и недостатки

Удобство использования шаблона Prototype заключается в том, что он позволяет системе сгенерировать копию готового к использованию объекта, поля которого имеют определенные, как правило, реальные, а не некие "абстрактные" значения, установленные конструктором. Схема использования шаблона Prototype представлена на рис. 1.5.

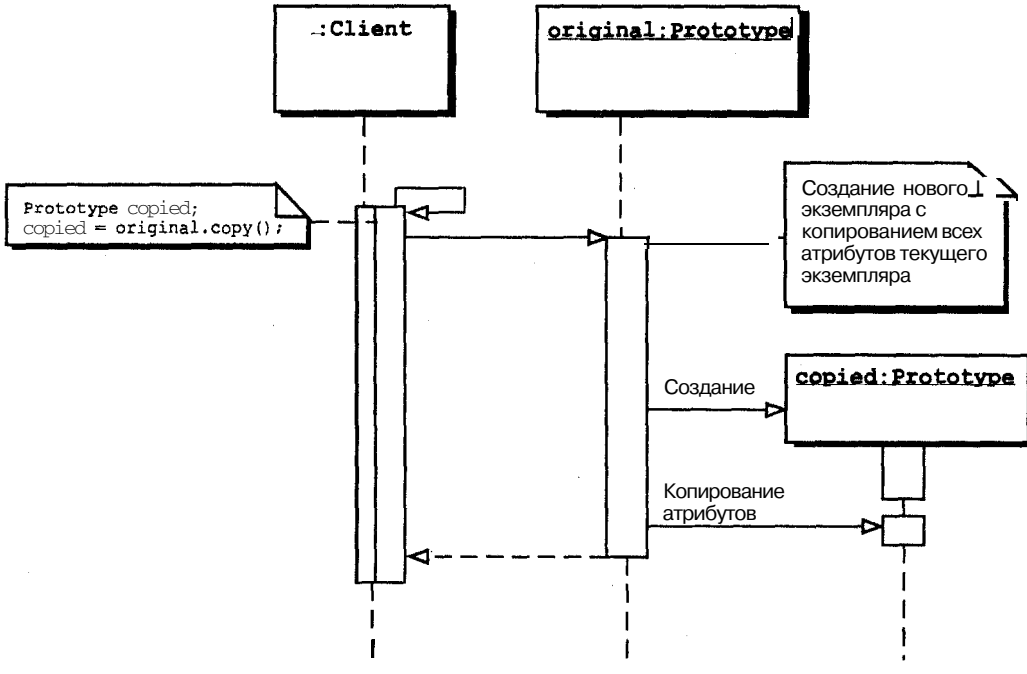


Рис. 1.5. Пример использования шаблона Prototype

Ключевое значение для данного шаблона имеет такое свойство, как уровень копирования.

- При создании частичной копии дублируются только элементы верхнего уровня класса. Это позволяет ускорить процесс копирования, но, с другой стороны, полученные дубликаты не во всех случаях могут быть полезными. Так как значения ссылок просто переписываются из оригинала в копию, соответствующие указатели продолжают ссылаться на те же объекты. Таким образом, объекты нижнего уровня одновременно используются несколькими копиями оригинального объекта, что приводит к изменению значений во всех копиях при внесении изменений в какой-либо одной из них.
- При полном копировании реплицируются не только атрибуты верхнего уровня, но и нижележащие объекты. Это, как правило, приводит к замедлению копирования, что для объектов с очень сложной структурой может выражаться в резком снижении производительности системы. Однако при таком копировании все копии не зависят ни друг от друга, ни от оригинального объекта.

Метод клонирования, присутствующий в классе Object, обеспечивает лишь одну форму копирования, поэтому в тех случаях, когда нужно обеспечить и частичное, и полное копирование, эта задача возлагается на разработчика.

Варианты

Данный шаблон может выполнятся в следующих вариантах.

- Копирующий конструктор. Одним из возможных вариантов реализации шаблона является оформление его в виде копирующего конструктора. Такой конструктор (листинг 1.17) получает в качестве параметра экземпляр своего класса и возвращает новую копию с теми же значениями полей, что и у объекта, переданного в качестве параметра.

Листинг 1.17. Копирующий конструктор

```
1. public class Prototype {
2.  private int someData;
3.   // Другие данные класса.
4.   public Prototype(Prototype original) {
5.     super () ;
6.      this.someData = original.someData;
1.      // Копирование остальных данных класса.
8.   }
9.   // Продолжение конструктора.
10.)
```

Примером такого класса является класс String, который позволяет создавать новый экземпляр класса String с помощью, например, такого вызова: new String("text");

Достоинство данного варианта состоит в том, что в тексте программы очень легко понять, когда создается новый экземпляр класса. Однако есть один нюанс — проще всего обеспечить лишь один вид копирования (либо частичное, либо пол-

ное). Конечно, это не столь существенно, так как можно написать конструктор, который будет выполнять оба вида копирования. Такой конструктор должен иметь два параметра: ссылку на копируемый объект и логическое значение, определяющее, какую копию (частичную или полную) нужно получить.

Недостатком копирующего конструктора можно назвать необходимость проверки полученной в качестве параметра ссылки, чтобы убедиться в том, что она не пустая. Обычно при реализации шаблона Prototype в данном варианте исходят из предположения о том, что вызываемый метод принадлежит существующему объекту.

- Метод клонирования. В языке программирования Java изначально встроен метод клонирования, определенный в классе `java.lang.Object`, который является суперклассом для всех классов Java. Для того чтобы в экземпляре класса можно было воспользоваться этим методом, класс должен реализовывать интерфейс `java.lang.Clonable`. Реализация этого интерфейса обеспечивает классу возможность клонирования. Так как метод `clone` объявлен в классе Object закрытым (`protected`), для обеспечения его вызова необходимо его перекрывать.

  В соответствии с Блохом, "методом `clone ()` нужно пользоваться очень взвешенно" [BlochOl]. Это объясняется тем, что даже если класс, как уже упоминалось, реализует интерфейс Clonable, это еще не гарантирует, что объект будет обладать свойством клонирования. Поскольку в интерфейсе Clonable отсутствует определение метода `clone`, возможны ситуации, когда этот метод, если он не был явно перекрыт, будет недоступен для вызова из класса. Еще одним недостатком этого метода является то, что он возвращает лишь объекты класса `Object`, что требует от программиста выполнять приведение типов перед использованием полученного клона.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Abstract Factory (стр. 26). Шаблон Abstract Factory может пользоваться шаблоном Prototype для создания новых объектов, аналогичных имеющемуся объекту.
- Factory Method (стр. 42). Шаблон Factory Method может пользоваться шаблоном Prototype, играя роль "заготовки" для новых объектов.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Prototype" на стр. 363 Приложения А.

Примером использования шаблона Prototype в данном примере является класс Address, который позволяет создавать новую адресную запись на основе уже имеющейся. Базовая функциональность шаблона определяется интерфейсом `Copyable` (листинг 1.18).

Листинг 1.18. Copyable.java

```
1. public interface Copyable{
2.   public Object copy();
3. )
```

Интерфейс CopyaIe содержит определение метода `copy` и обеспечивает определение операции копирования для любого класса, который реализует этот интерфейс. В данном примере мы рассмотрим частичное копирование, т.е. копирование, при котором объектные ссылки оригинального объекта просто дублируются в его копии.

В листинге 1.19 продемонстрировано важное свойство операции копирования, состоящее в том, что далеко не во всех ситуациях обязательно копировать все поля без исключения. В данном случае в новый объект не копируется информация о типе адреса, так как пользователь, скорее всего захочет его изменить, используя графический интерфейс PIM-приложения.

Листинг 1.19. Address.java

```
1. public class Address implements Copyable{
2.   private String type;
3.   private String street;
4.   private String city;
5.   private String state;
6.   private String zipCode;
7.   public static final String EOL_STRING =
8.      System.getProperty("line.separator");
9.   public static final String COMMA = ",";
10.  public static final String HOME = "home";
11.  public static final String WORK = "work";
12.
13.  public Address(String initType, String initStreet,
14.     String initCity, String initState, String initZip){
15.       type = initType;
16.       street = initStreet;
17.       city = initCity;
18.       state = initState;
19.       zipCode = initZip;
20.  }
21.
22.  public Address(String initStreet, String initCity,
23.     String initState, String initZip)(
24.       this(WORK, initStreet, initCity, initState, initZip);
25.  }
26.  public Address(String initType))
27.     type = initType;
28.  }
29.  public Address(){ )
30.
31.  public String getType(){ return type; }
32.  public String getStreet(){ return street; }
33.  public String getCity(){ return city; }
```

```
34. public String getState() { return state; }
35. public String getZipCode() { return zipCode; }
36.
37. public void setType(String newType) { type = newType; }
38. public void setStreet(String newStreet) { street = newStreet; }
39. public void setCity(String newCity) { city = newCity; }
40. public void setState(String newState) { state = newState; }
41. public void setZipCode(String newZip) { zipCode = newZip; }
42.
43. public Object copy(){
44.   return new Address(street, city, state, zipCode);
45. }
46.
47. public String toString(){
48.    return "\t" + street + COMMA + " " + EOL_STRING +
49.      "\t" + city + COMMA + " " + state + " " + zipCode;
50.   }
51.}
```

Singleton

Свойства шаблона

Тип: производящий шаблон
Уровень: объект

Назначение

Обеспечивает наличие в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру.

Представление

Предположим, что нам понадобился некий глобальный объект, т.е. такой объект, доступ к которому можно было бы осуществить из любой точки приложения, но при этом необходимо, чтобы он создавался только один раз. Иными словами, к этому объекту должны иметь доступ все элементы приложения, но работать они должны с одним и тем же экземпляром.

Примером такого объекта может быть хронологический список (history list), в котором хранится информация о всех действиях пользователя, которые он предпринимал, работая с приложением. Объект `HistoryList` по определению должен быть доступным для всех элементов приложения, чтобы они могли либо заносить в него сведения об очередной операции, выполненной пользователем, либо извлекать из него данные о последней операции для ее отмены.

Один из возможных методов решения этой задачи состоит в создании глобального объекта в главном модуле приложения с последующей передачей ссылки на этот объект всем другим объектам, которым это необходимо. Однако довольно трудно, приступая к разработке приложения, правильно определить способ передачи ссылки, который бы подходил для всех объектов, равно как и заранее предугадать, каким именно элементам приложения понадобится этот объект. Еще одним недостатком данного

решения является невозможность воспрепятствовать другим объектам создавать дополнительные экземпляры глобального объекта (в нашем случае — `HistoryList`).

Существует и другой способ получения глобальных значений, основанный на применении статических переменных. Это позволяет приложению обращаться напрямую к нескольким специальным статическим объектам, заключенным внутри некоторого класса.

Но данный подход также имеет ряд недостатков.

- Статический объект — это не самое лучшее решение, так как он создается во время загрузки класса, что лишает разработчика возможности передачи ему данных перед созданием экземпляра.
- Разработчик не может контролировать доступ к статическому объекту, который объявлен общедоступным.
- Если разработчик решит, что вместо одного объекта ему понадобится, скажем, тройка таких объектов, ему придется практически заново переписать весь код клиентской части приложения.

В подобной ситуации очень полезным становится шаблон Singleton, который обеспечивает удобный доступ всех элементов приложения к глобальному объекту.

Область применения

Шаблон Singleton применяется в тех случаях, когда нужно, имея лишь один экземпляр класса, обеспечить доступ к нему всем элементам приложения.

Описание

Шаблон Singleton гарантирует на уровне JVM (Java Virtual Machine — виртуальная машина Java), что в приложении не будет создано более одного экземпляра класса. Для того чтобы обеспечить контроль над созданием экземпляров класса, достаточно сделать конструктор закрытым (private).

Однако тут возникает небольшая проблема: как же теперь создать хотя бы один экземпляр, если эта операция недоступна? Эта проблема решается с помощью предоставления специального статического метода доступа (`getInstance()`). Данный метод создает единственный экземпляр (если, конечно, он уже не существует) и возвращает вызвавшему его элементу ссылку на созданный объект-одиночку. Эта ссылка в свою очередь сохраняется как статический закрытый атрибут класса полученного объекта для последующего использования при вызовах.

Хотя объект-одиночку можно создать с помощью специального метода доступа, все же чаще всего он создается во время загрузки класса. Отложенный вызов конструктора оправдано лишь в тех случаях, когда перед созданием экземпляра необходимо выполнить некоторые настроечные инициализационные операции.

Примером объекта-одиночки может служить Президент Соединенных Штатов Америки. Действительно, в США в любой момент времени может быть лишь один действующий Президент. Когда Президент Российской Федерации снимает трубку на аппарате прямой связи, он ожидает, что его собеседником на другом конце провода будет именно Президент США.

Реализация

Диаграмма классов шаблона Singleton представлена на рис. 1.6.

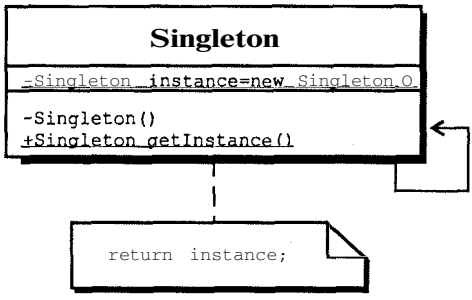


Рис. *1.6.* Диаграмма класса Singleton

При реализации шаблона Singleton используется класс Singleton. В этом классе определяется закрытый конструктор, имеется закрытая статическая переменная, в которой хранится ссылка на единственный экземпляр данного класса, а также определен статический метод доступа, возвращающий ссылку на этот экземпляр.

Остальные элементы класса Singleton не отличаются от элементов других классов. Статический метод доступа может реализоваться таким образом, чтобы он мог принимать решение о том, какой экземпляр создавать, базируясь на свойствах системы или значениях переданных ему параметров (подробнее об этом говорится в подразделе "Варианты" текущего раздела).

Достоинства и недостатки

Шаблону Singleton присущи следующие достоинства и недостатки.

- Единственным классом, который может создать экземпляр класса Singleton, является сам класс Singleton, причем создать его можно, лишь пользуясь предоставленным статическим методом.
- Разработчик избавлен от необходимости передавать ссылку на объект-одиночку всем объектам, которым требуется получить к нему доступ.
- С другой стороны, в зависимости от реализации шаблон Singleton может представлять собой проблему для многопоточных приложений. В таких приложениях необходимо с повышенным вниманием следить за процессом инициализации объекта-одиночки, так как в некоторых ситуациях приложение может войти в состояние "войны потоков".

Варианты

Данный шаблон может выполнятся в следующих вариантах.

- Одним из свойств шаблона Singleton, которое часто остается незамеченным для разработчиков, является возможность иметь несколько экземпляров внутри класса. Достоинство такого варианта состоит в том, что, не переписывая остальную часть приложения, можно предоставить отдельным объектам специальные методы для получения доступа к дополнительным экземплярам.
- Метод доступа шаблона Singleton может открыть целый набор экземпляров, каждый из которых может относиться к другому подтипу. Метод доступа может определять прямо во время выполнения, экземпляр какого именно подтипа нужно вернуть. На первый взгляд такая возможность может показаться избыточной, но она, как правило, оказывается весьма полезной при использовании динамической загрузки классов. Система, в которой используется шаблон Singleton, остается неизменной, но при этом меняется нужным образом реализация объекта-одиночки.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Abstract Factory (стр.26).
- Builder (стр. 34).
- Prototype (стр. 48).

Пример

Пользователям приложения, без сомнения, понадобится функция отмены предыдущих команд. Для того чтобы обеспечить поддержку в приложении этой функциональности, необходимо добавить в него хронологический список операций. Соответствующий объект должен быть доступен из любого другого объекта PIM-приложения, причем все объекты должны работать с одним и тем же единственным экземпляром класса. Таким образом, хронологический список — это практически идеальный кандидат на применение шаблона Singleton (листинг 1.20).

Листинг 1.20. Singleton.java

```
1. import java.util.ArrayList;
2. import java.util.Collections;
3. import java.util.List;
4. public class HistoryList{
5.   private List history = Collections.synchronizedList(new ArrayList());
6.   private static HistoryList instance = new HistoryList();
7.
8.   private HistoryList(){ }
9.
10.  public static HistoryList get Instance(){
```

```
11.    return instance;
12. }
13.
14. public void addCommand(String command){
15.    history.add(command);
16. }
17.
18. public Object undoCommand(){
19.    return history.remove(history.size() - 1);
20. }
21.
22. public String toString(){
23.    StringBuffer result = new StringBuffer();
24.    for (int i = 0 ; i < history.size(); i++){
25.     result.append(" " ) ;
26.     result.append(history.get(i) ) ;
27.       result.append("\n");
28.     }
29.     return result.toString();
30. }
31.}
```

Класс `HistoryList` содержит статическую переменную, в которой хранится ссылка на собственный экземпляр класса, а также имеет закрытый конструктор и метод `getInstance`, предоставляющий ссылку на единственный объект хронологического списка всем элементам приложения. Еще одна переменная, `history`, представляет собой объект класса `List`, который используется для отслеживания текстового представления команд. Кроме того, у класса HistoryList имеется два метода (addCommand и undoCommand), предназначенные для обеспечения, соответственно, добавления команд к списку и удаления их из списка.

ПОВЕДЕНЧЕСКИЕ ШАБЛОНЫ

Глава 2

Введение

Поведенческие шаблоны (behavioral patterns) применяются для передачи управления в системе. Существуют методы организации управления, применение которых позволяет добиться значительного повышения как эффективности системы, так и удобства ее эксплуатации. Поведенческие шаблоны представляют собой квинтэссенцию проверенных на практике методов и обеспечивают понятные и простые в применении эвристические способы организации управления.

В данной главе рассматриваются следующие поведенческие шаблоны.

- *Chain of Responsibility.* Предназначен для организации в системе уровней ответственности. Использование этого шаблона позволяет установить, должно ли сообщение обрабатываться на том уровне, где оно было получено, или же оно должно передаваться для обработки другому объекту.
- *Command.* Обеспечивает обработку команды в виде объекта, что позволяет сохранять ее, передавать в качестве параметра методам, а также возвращать ее в виде результата, как и любой другой объект.
- *Interpreter.* Определяет интерпретатор некоторого языка.
- *Iterator.* Предоставляет единый метод последовательного доступа к элементам коллекции, не зависящий от самой коллекции и никак с ней не связанный.
- *Mediator.* Предназначен для упрощения взаимодействия объектов системы путем создания специального объекта, который управляет распределением сообщений между остальными объектами.
- *Memento.* Сохраняет "моментальный список" состояния объекта, позволяющий такому объекту вернуться к исходному состоянию, не раскрывая своего содержимого внешнему миру.

- *Observer.* Предоставляет компоненту возможность гибкой рассылки сообщений интересующим его получателям.
- *State.* Обеспечивает изменение поведения объекта во время выполнения программы.
- *Strategy.* Предназначен для определения группы классов, которые представляют собой набор возможных вариантов поведения. Это дает возможность гибко подключать те или иные наборы вариантов поведения во время работы приложения, меняя его функциональность "на ходу".
- *Visitor.* Обеспечивает простой и удобный в эксплуатации способ выполнения тех или иных операций для определенного семейства классов. Это достигается путем централизации с помощью данного шаблона возможных вариантов поведения, что позволяет модифицировать или расширять их, не затрагивая классы, на которые распространяются эти варианты поведения.
- *Template Method.* Предоставляет метод, который позволяет подклассам перекрывать части метода, не прибегая к их переписыванию.

Примечание

К поведенческим шаблонам можно отнести также и шаблон MVC (Model-View-Controller). Однако, учитывая его влияние на всю систему в целом, особенно в контексте рекомендаций J2EE по спецификациям сервлетов и JSP, мы поместили его в главе 4, "Системные шаблоны" (стр. 219).

Chain of Responsibility

Свойства шаблона

Тип: поведенческий шаблон
Уровень: компонент

Назначение

Предназначен для организации в системе уровней ответственности. Использование этого шаблона позволяет установить, должно ли сообщение обрабатываться на том уровне, где оно было получено, или же оно должно передаваться для обработки другому объекту.

Представление

PIM-приложение может управлять не только контактами, но и проектами. Список проектов можно представить в виде древовидной структуры объектов, каждый из которых соответствует одной задаче. В "корне" дерева также находится одна задача, которая представляет весь проект в целом. Эта базовая задача связана с набором подза-

дач, каждая из которых имеет собственные наборы подзадач и т.д. Данный подход позволяет разделить крупный проект на все более и более детализированные наборы взаимосвязанных целей. Это позволяет пользователю прибегнуть к групповым операциям, соответствующим тем или иным целям, как показано в приведенном ниже примере.

- Проект (базовая задача): завладеть страной.
  - Подзадача: воспользоваться счастливым случаем.
    - Подзадача: попробовать выиграть в лотерею.
  - Подзадача: выяснить, где лучше климат — в Тихом океане или в Атлантическом.
  - Подзадача: найти остров, выставленный на продажу.
    - Подзадача: выяснить, продаются ли острова на аукционе E-Bay.
  - Подзадача: изучить процедуры ООН, относящиеся к образованию новых государств.
  - Подзадача: придумать название своей страны.

Как лучше всего управлять информацией, имеющей подобную структуру? Один из возможных подходов состоит в том, чтобы распределить задачи по уровням ответственности. Но, как лучше организовать делегирование одной группы задач одному человеку, другой — другому и т.д.?

Возможным решением является определение специального атрибута для каждой задачи, определяющего его владельца. Когда владелец задачи меняется, обновляется информация о владельце и для всех подзадач данной задачи. Но это решение не очень удачно, так как требует хранить дополнительную информацию о владельце для каждой задачи, что усложняет работу пользователя.

Другое решение состоит в создании одного или нескольких централизованных объектов, которые хранят информацию о владельцах задач. Хотя это решение более эффективно с точки зрения управления памятью, оно также требует дополнительных затрат по управлению связями между задачами и централизованным хранилищем данных об их владельцах.

А что если использовать для управления владельцами само дерево задач? Определим новый метод класса Task, например, как `getOwner` и свяжем его с атрибутом owner. При вызове этот метод будет проверять, имеется ли у задачи владелец (т.е. отлично ли значение атрибута owner от нуля). Если владелец уже определен, метод возвращает его имя, а если нет — объект класса Task вызывает метод getOwner вышележащего объекта. Данное решение гораздо менее трудоемкое, чем первые два, и при этом эффективно с точки зрения использования памяти. Пользователю требуется указывать владельцев задач лишь один раз, так как они автоматически назначаются таковыми классом `Task` для всех подзадач своей задачи. Это обеспечивается путем делегирования вызовов метода getOwner нижележащих классов к вышележащим до тех пор, пока не будет получена запрашиваемая информация о владельце задачи. Данное решение иллюстрирует типичный случай применения шаблона Chain of Responsibility.

Область применения

Шаблон Chain of Responsibility рекомендуется использовать в следующих случаях.

- В системе имеется группа объектов, любой из которых может обрабатывать сообщения определенного типа.
- Полученное сообщение должно быть обработано хотя бы одним из объектов системы.
- Сообщения обрабатываются по схеме "обработай сам или перешли другому". Иными словами, некоторые сообщения обрабатываются на том уровне, где они были получены, тогда как другие пересылаются каким-то объектам иного уровня.

Описание

Когда в системе, построенной на принципах объектно-ориентированного программирования, выполняются какие-либо операции, они часто представляются в виде *событий* (event) или *сообщений* (message). Сообщение может оформляться в виде подлежащего вызову метода или же в виде отдельного объекта системы. Обычно сообщение направляется другому объекту, который определенным образом реагирует на него или, как принято говорить, *обрабатывает* (handle) это сообщение.

В простейших случаях сообщение обрабатывается тем объектом, который его сгенерировал. Например, строка ввода может и генерировать сообщения в ответ на определенные действия пользователя (такие как ввод текста с клавиатуры), и обрабатывать их (отображая введенный текст в своем поле).

В более сложных случаях обработка сообщений может идти по многоступенчатой схеме. Например, сообщение, требующее изменения внешнего вида или размера компонента пользовательского интерфейса, может обрабатываться на разных уровнях. Если требуется изменить режим выравнивания текста в поле, то такое сообщение может обработать и сам компонент. Но если требуется изменить режим выравнивания самого текстового поля, сообщение скорее всего будет перенаправлено какому-то объекту более высокого уровня, содержащему это поле, такому, например, как панель или окно. Эту модель вполне можно реализовать с помощью шаблона Chain of Responsibility.

Шаблон Chain of Responsibility позволяет построить схему уровней обработки сообщений. Если объект не может обработать какое-то сообщение, он передает его другому объекту в соответствии с установленной схемой. Часто шаблон Chain of Responsibility реализуется в виде модели "родительский объект — дочерний объект" или "контейнер — содержимое". В соответствии с такой моделью сообщение, не обработанное дочерним объектом, пересылается родительскому объекту, затем, возможно, его родительскому объекту и т.д. до тех пор, пока не будет найден объект, имеющий нужный обработчик.

Шаблон Chain of Responsibility хорошо подходит для самых различных операций с системой GUI, построенной на объектно-ориентированных принципах. Поддержка функций справочной системы GUI, изменение размера, форматирование и размещение компонентов GUI — все эти задачи могут быть решены благодаря использованию данного шаблона. В бизнес-моделях шаблон Chain of Responsibility иногда используется в виде

модели "целое — частное". Например, объект, представляющий собой отдельную строку заказа, может отправить сообщение объекту, представляющему весь заказ в целом, запрашивая выполнение какой-то операции.

Работу шаблона Chain of Responsibility можно проиллюстрировать следующим примером. Рассмотрим, как в организации проходит служебная записка с запросом на командировку. Обычно такой запрос должен быть завизирован менеджером соответствующего уровня. Если в записке идет речь о местной командировке на несколько часов, достаточно визы непосредственного руководителя работника. Если же работник просит о направлении его в командировку в другую страну, то скорее всего его служебная записка будет перемещаться по иерархическим уровням организации до тех пор, пока не дойдет до менеджера с самым высоким уровнем полномочий.

Схема работы шаблона Chain of Responsibility показана на рис. 2.1.

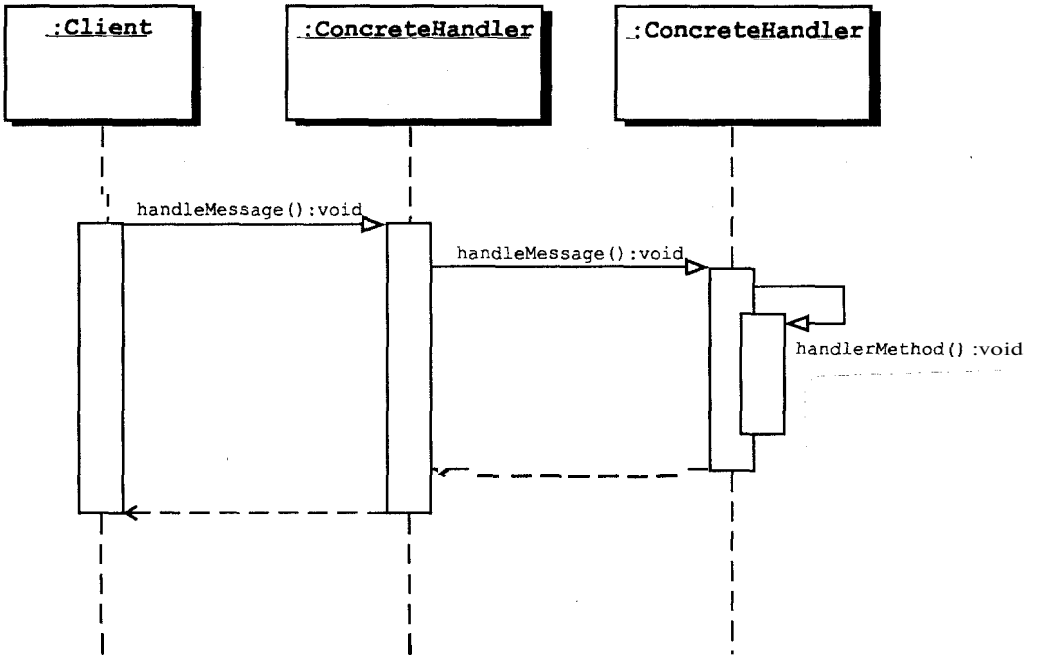


Рис. 2.1. Схема работы шаблона Chain of Responsibility

Реализация

Диаграмма классов шаблона Chain of Responsibility представлена на рис. 2.2.

При реализации шаблона Chain of Responsibility обычно используются следующие классы.

- Handler. Интерфейс, который определяет метод, используемый для передачи сообщения следующему обработчику. Сообщение — это обычно просто вызов метода. В тех случаях, когда в сообщении нужно инкапсулировать больше данных, его роль может играть и отдельный объект.

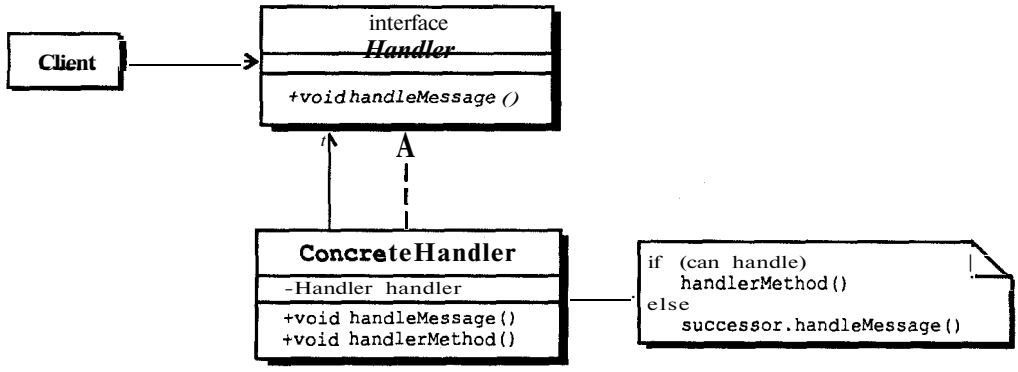


Рис. *2.2.* Диаграмма классов шаблона Chain of Responsibility

- ConcreteHandler. Класс, реализующий интерфейс Handler. Именно в этом классе хранится ссылка на следующий объект Handler. Эта ссылка устанавливается либо в конструкторе класса, либо с помощью вызова специального метода. Реализация метода handleMessage может определять, как обрабатывать метод и вызывать `handleMethod`, либо как пересылать сообщение следующему объекту Handler, или и то, и другое.

Достоинства и недостатки

Шаблон Chain of Responsibility привносит в приложение большую гибкость во всем, что касается обработки событий, так как он управляет этим довольно сложным процессом, распределяя ответственность между несколькими более простыми элементами. Он позволяет действовать некоторому подмножеству классов как единому целому, поскольку события, сгенерированные одним классом, могут перенаправляться для обработки другим классам подмножества.

Конечно, гибкость, предоставляемая данным шаблоном, не дается без труда: шаблон Chain of Responsibility трудно разработать, протестировать и отладить. По мере того, как система уровней, пересылающих друг другу сообщения, становится сложнее, разработчику приходится все внимательнее следить за тем, чтобы события обрабатывались корректно.

Неудачное планирование различных средств пересылки сообщений может приводить к появлению "пропавших сообщений" (сообщений, для которых не существует обработчика и которые, таким образом, никогда не будут обработаны), а также к "замусориванию" коммуникационных механизмов. Под "замусориванием" понимается слишком высокий поток сообщений и чрезмерно большое количество этапов пересылки. Если за короткий промежуток времени в системе генерируется слишком много сообщений и их приходится пересылать по несколько раз, пока не освободиться обработчик, это может привести к существенному снижению производительности системы.

Варианты

Шаблон Chain of Responsibility можно адаптировать к потребностям приложения несколькими способами. Основные подходы базируются либо на выбранных стратегиях обработки, либо на стратегиях пересылки.

Стратегии обработки (handling strategies) фокусируются на том, как реализовано поведение обработчика. Среди многих вариантов можно выделить следующие.

- *Используемый по умолчанию обработчик.* В некоторых реализациях устанавливается базовый обработчик, который используется всеми уровнями как обработчик по умолчанию. Этот подход обычно применяется только в тех случаях, когда в системе отсутствует явно определенный класс пересылки. Используемый по умолчанию обработчик особенно полезен для решения проблемы пропавших сообщений, упоминавшейся в приведенном выше подразделе "Достоинства и недостатки" данного раздела.
- *Расширение обработчика.* В этом варианте обработчики, расположенные на каждом последующем уровне, добавляют свою функциональность к функциональности базового обработчика. Например, данный вариант может оказаться полезным в таком случае, как реализация ведения журналов.
- *Динамические обработчики.* В некоторых реализациях шаблона Chain of Responsibility разрешается изменять структуру подсистемы пересылки сообщений во время выполнения программы. Определив в каждом классе каждого уровня специальный метод, можно определять и модифицировать схему передачи сообщений во время работы приложения (естественно, со всеми последствиями, вытекающими из такого усложнения системы).

Разные виды *стратегии пересылки* (forwarding strategies) определяют различные подходы к обработке или пересылке сообщений, генерируемых компонентом.

- *Обработка по умолчанию.* Обрабатываются все сообщения, кроме тех, пересылка которых определена явным образом.
- *Пересылка по умолчанию.* Пересылаются все сообщения, кроме тех, обработка которых определена явным образом.
- *Пересылка используемому по умолчанию обработчику.* В данном, несколько усложненном по сравнению с базовым, варианте используется обработчик по умолчанию. Любое сообщение, для которого явным образом не указано, что оно должно быть обработано на уровне заданного компонента или переслано заданному обработчику, пересылается обработчику, используемому по умолчанию.
- *Игнорирование по умолчанию.* Отбрасывается любое сообщение, для которого не было определено явным образом, что оно должно быть обработанным или пересланным. Данный вариант полезен в тех случаях, когда классы на каком-либо уровне генерируют события, которые не используются в приложении, позволяя таким образом, уменьшить эффект "замусоривания". Однако к его применению необходимо подходить осторожно, чтобы не столкнутся с безвозвратной потерей важных для системы сообщений.

Родственные шаблоны

К родственным можно отнести структурный шаблон Composite (стр. 171). Шаблон Chain of Responsibility часто используется именно с шаблоном Composite. При их совместном использовании последний обеспечивает поддержку древовидной структуры и базового механизма передачи сообщений, тогда как шаблон Chain of Responsibility предоставляет правила, регулирующие передачу сообщений.

Кроме того, шаблон Composite организует пересылку сообщений "вниз" по дереву (от корня к ветвям), тогда как шаблон Chain of Responsibility обычно обеспечивает их пересылку "вверх" по дереву (от ветвей к корню).

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Chain of Responsibility" на стр. 370 Приложения А.

PIM-приложение может управлять не только контактами, но и проектами. В данном примере показано, как использовать шаблон Chain of Responsibility для получения информации, хранящейся в иерархическом списке проекта.

Интерфейс `ProjectItem` (листинг 2.1) определяет общие методы для любого элемента, который может быть частью проекта.

Листинг 2.1. `ProjectItem.java`

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public interface ProjectItem extends Serializable(
4.   public static final String EOL_STRING =
   System.getProperty("line.separator");
5.    public ProjectItem getParent();
6.    public Contract getOwner();
7.    public String getDetails();
8.    public ArrayList getProjectItems();
9. }
```

Интерфейс определяет методы getParent, getOwner, getDetails и getProjectItems. В рассматриваемом примере интерфейс ProjectItem реализуется двумя классами — Project и Task. Класс `Project` (листинг 2.2) является базой проекта, поэтому его метод getParent возвращает пустой указатель (null). Методы getOwner и getDetails возвращают, соответственно, владельца и описание всего проекта в целом, а метод `getProjectItems` — ссылки на все элементы проекта нижележащего уровня.

Листинг 2.2. Project.java

```
1. import java.util.ArrayList;
2. public class Project implements ProjectItem{
3.   private String name;
4.   private Contact owner;
5.   private String details;
6.   private ArrayList projectItems = new ArrayList();
7.
8.  public Project () { }
9.   public Project (String newName, String newDetails, Contact newOwner){
10.     name = newName;
11.     Owner = newOwner;
12.     details = newDetails;
13. }
14.
15.  public String getName(){ return name; }
16.  public String getDetails () { return details; }
17.  public Contact getOwner(){ return owner; }
18.  public ProjectItem getParent(){ return null; }
19.  public ArrayList getProjectItems(){ return projectItems; }
20.
21.  public void setName(String newName){ name = newName; }
22.  public void setOwner(Contact newOwner){ owner = newOwner; }
23.  public void setDetails(String newDetails){ details = newDetails; }
24.
25.  public void addProjectItem(ProjectItem element){
26.     if (!projectItems.contains(element)){
27.       projectItems.add(element);
28.     }
29. }
30.
31.  public void removeProjectItem(ProjectItem element){
32.     projectItems.remove(element);
33. }
34.
35.  public String toString(){
36.     return name;
37. }
38.}
```

Класс Task (листинг 2.3) представляет собой некоторую отдельную задачу в рамках проекта. Подобно классу Project, класс Task также хранит коллекцию подзадач, ссылку на которую возвращает его метод getProjectItems. Метод getParent класса Task возвращает ссылка на родительский объект, который может быть экземпляром класса Task или Project.

Листинг 2.3. Task.java

```
1. import java.util.ArrayList;
2. import java.util.ListIterator;
3. public class Task implements ProjectItem{
4.   private String name;
5.   private ArrayList projectItems = new ArrayList();
6.   private Contact owner;
7.   private String details;
8.   private ProjectItem parent;
9.   private boolean primaryTask;
```

```
10.
11.  public Task(ProjectItem newParent){
12.    this(newParent, "", "",null, false);
13.  }
14.  public Task(ProjectItem newParent, String newName,
15.    String newDetails, Contact newOwner, boolean newPrimaryTask){
16.      parent = newParent;
17.      name = newName;
18.      owner = newOwner;
19.      details = newDetails;
20.      primaryTask = newPrimaryTask;
21.  }
22.
23.  public Contact getOwner(){
24.    if (owner == null){
25.      return parent.getOwner()
26.    }
27.    else{
28.      return owner;
29.    }
30.  }
31.
32.  public String getDetails(){
33.    if (primaryTask){
34.      return details;
35.    }
36.    elsef
37.     return parent.getDetails()  + EOL_STRING + "\t" + details;
38.    )
39.  }
40.
41.  public String getName(){ return name; }
42.  public ArrayList getProjectItems(){ return projectItems; }
43.  public Projectltem getParent()f return parent;}
44.  public boolean isPrimaryTask(){ return primaryTask; }
45.
46.  public void setName(String newName)) name = newName; )
47.  public void setOwner(Contact newOwner){ owner = newOwner; }
48.  public void setParent(ProjectItem newParent)( parent = newParent; }
49.  public void setPrimaryTask(boolean newPrimaryTask){ primaryTask =
   newPrimaryTask; }
50.  public void setDetails(String newDetails){ details = newDetails; }
51.
52.  public void addProjectItem(ProjectItem element){
53.    if (!projectItems.contains(element)){
54.      projectItems.add(element);
55.    }
56.  }
57.
58.  public void removeProjectItem(ProjectItem element)}
59.    projectItems.remove(element);
60.  }
61.
62.  public String toString(){
63.    return name;
64.  }
65.}
```

Поведение, характерное для шаблона Chain of Responsibility, проявляется в методах getOwner и getDetails класса Task. Так, метод getOwner этого класса возвращает либо значение атрибута владельца своего класса (если оно отлично от нуля), ли-

бо значение, предоставленное родительским классом. Если родительским классом является класс `Task` и его атрибут владельца также не установлен, вызов метода передается следующему родителю до тех пор, пока не будет найдено отличное от нуля значение или пока не будет вызван аналогичный метод класса `Project`. Это позволяет без особого труда создать группу задач, представленных экземплярами класса `Task`, владельцем которых будет назначено одно лицо, отвечающее как за исполнение главной задачи `Task`, так и всех ее подзадач.

Метод `getDetails` также является примером поведения, типичного для шаблона Chain of Responsibility, хотя оно проявляется несколько иначе. Данный метод вызывает метод `getDetails` каждого из родителей до тех пор, пока не достигнет класса `Task` или `Project`, идентифицированного как терминальный узел. Это означает, что метод `getDetails` возвращает набор объектов класса `String`, совокупность которых представляет описание конкретной задачи с глубиной детализации, соответствующей точке вызова `getDetails`.

Command

Также известен как Action, Transaction

Свойства шаблона

Тип: поведенческий шаблон
Уровень: объект

Назначение

Обеспечивает обработку команды в виде объекта, что позволяет сохранять ее, передавать в качестве параметра методам, а также возвращать ее в виде результата, как и любой другой объект.

Представление

Когда пользователь выполняет какую-либо операцию, приложение должно знать, где взять соответствующие данные и что с ними делать. Обычно приложению известны все варианты действий, предложенных пользователю, и где находится соответствующая им логика, которая жестко "прошивается" в приложении на этапе разработки. Когда пользователь выбирает то или иное действие, приложение определяет, что нужно сделать, выбирает требуемые данные, а затем вызывает необходимые методы.

Конечно, ни один программист *никогда* не допускает ошибок, однако, пользователи — это обычные люди, которые иногда ошибаются. Именно поэтому многие современные приложения позволяют пользователям отменять несколько последних операций, выполненных после определенной контрольной операции, например, после последнего сохранения.

Для того чтобы добавить такую возможность в свое приложение, разработчик должен создать объект, известный как хронологический список. Этот объект содер-

жит список всех выполненных пользователем операций, список всех данных, которые требовались для их выполнения, а также сведения о состоянии, в котором находилось приложение до выполнения указанных операций. Однако из-за избыточности хранимых данных уже после двух-трех операций хронологический список может занимать больше памяти, чем все приложение.

Поэтому имеет смысл объединить все, что описывает выполняемую пользователем операцию, в один объект, соответствующий шаблону Command. В таком объекте содержатся как данные, необходимые для выполнения операции, так и логика этой операции. Теперь приложению для того, чтобы выполнить команду, достаточно вызвать метод `execute` объекта Command. Приложению больше не нужно хранить сведения обо всех доступных для пользователя вариантах выбора. Кроме того, при таком подходе в значительной степени упрощается добавление в приложение новых операций.

Область применения

Шаблон Command рекомендуется использовать в следующих случаях.

- Необходимо обеспечить поддержку таких операций, как отмена, ведение журнала и (или) операций с транзакциями.
- Необходимо обеспечить постановку команд в очередь и их выполнение в заданное время.
- Требуется отделить источник запроса от объекта, отвечающего на запрос.

Описание

В приложении, в котором не используется шаблон Command, нужно предусмотреть в классе обработчика наличие отдельного метода для каждого типа события. Иными словами, чтобы выполнить запрашиваемую операцию, обработчик должен иметь всю информацию. Добавление новых операций в таком случае потребует добавления новых методов в класс обработчика.

Шаблон Command для выполнения определенной операции или определенного запроса инкапсулирует в себе все данные и необходимую функциональность. Он обеспечивает разделение между тем, *когда* нужно выполнить операцию, и тем, *как* ее нужно выполнить.

Приложение, использующее шаблон Command, определяет источник (например, GUI), получателя (объект, который задействуется запросом) и команду (Listener). Команде передается ссылка на получателя, а источнику — ссылка на команду. В данном примере, когда пользователь щелкнет на кнопке GUI, создастся метод `execute` или `listener`объекта команды (рис. 2.3).

Созданный объект команды передается объекту, который должен вызвать эту команду (invoker). Этот объект реализует интерфейс Command. В самом простом случае в интерфейсе определяется лишь метод выполнения `execute`. Реализующие классы сохраняют ссылку на получателя в специальной переменной экземпляра. Когда вызывается метод выполнения, Command вызывает метод `doAction` класса Receiver. Класс Command может вызывать несколько разных методов класса Receiver.

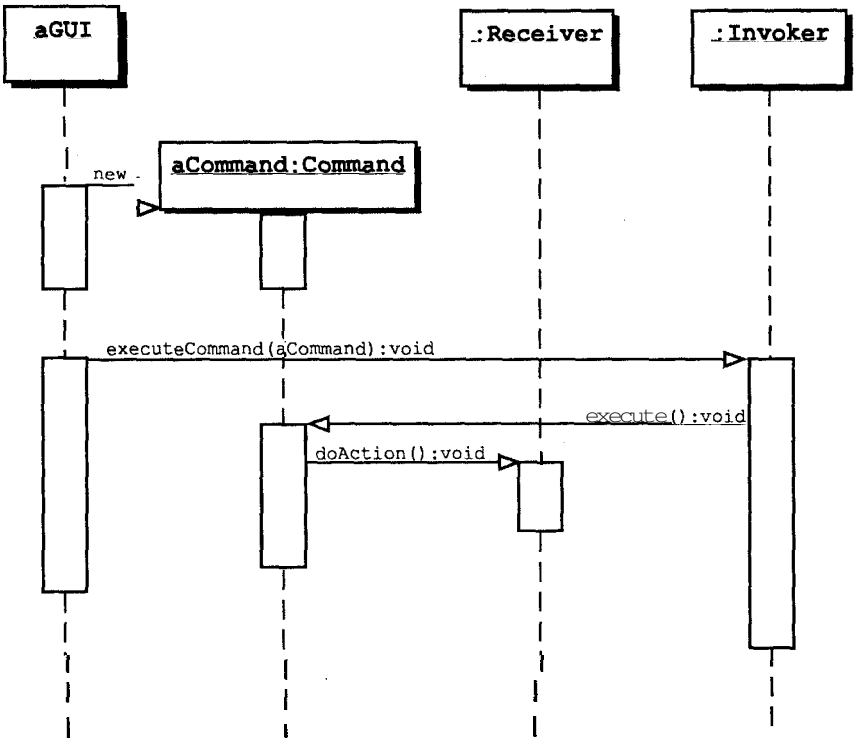


Рис. *2.3.* Схема работы шаблона Command при вызове команды

Реализация

Диаграмма классов шаблона Command представлена на рис. 2.4.

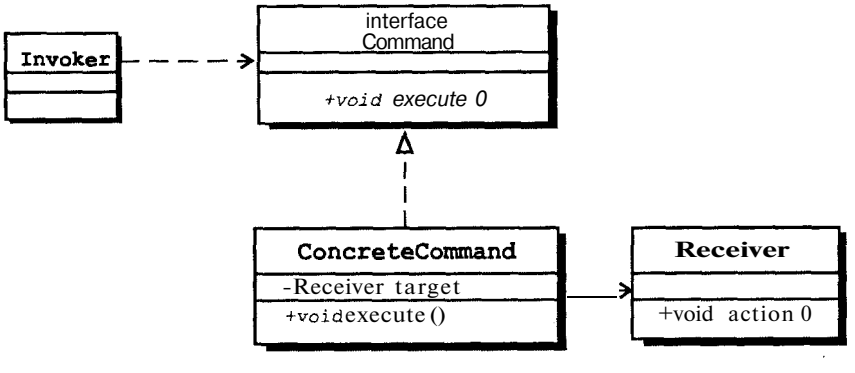


Рис. *2.4.* Диаграмма классов шаблона Command

При реализации шаблона Command обычно используются следующие классы.

- Command. Интерфейс, определяющий методы, которые использует класс Invoker.
- Invoker. Класс, вызывающий метод выполнения объекта Command.
- Receiver. Объект, которому предназначен объект Command и который выполняет указанный в команде запрос, для чего он обладает всей необходимой информацией.
- `ConcreteCommand`. Реализация интерфейса Command. В нем хранится ссылка на предназначенный для выполнения конкретной команды объект класса Receiver. Когда вызывается метод выполнения, ConcreteCommand вызывает один или несколько методов объекта Receiver.

При реализации шаблона Command необходимо принять решение о способе обработки вызовов. Такая обработка может выполняться одним из следующих способов.

- Класс, который реализует интерфейс Command, может быть лишь связующим звеном между вызывающим и вызываемым объектами и просто выполнять пересылку вызовов. Такой подход позволяет обеспечить простоту класса `ConcreteCommand`.
- Класс ConcreteCommand может быть получателем и обрабатывать все запросы самостоятельно. Данный подход полезен в тех случаях, когда для определенного запроса не существует конкретного получателя.

Конечно, нет никаких препятствий в том, чтобы объединить эти два подхода и обрабатывать одну часть запросов классом ConcreteCommand, а другую часть пересылать другим классам.

Достоинства и недостатки

Шаблон Command увеличивает гибкость приложения следующими способами.

- Отделение источника события от объекта, который знает, как нужно выполнять запрашиваемую операцию.
- Совместное использование экземпляров Command разными объектами.
- Обеспечение возможности замены объектов Command и (или) Receiver во время выполнения.
- Представление команд в виде обычных объектов, обладающих всеми присущими объектам свойствами.
- Упрощение добавления новых команд — достаточно лишь написать еще одну реализацию интерфейса и добавить ее к приложению, не меняя обработчика.

Варианты

Шаблон Command может реализовываться в следующих вариантах.

- *Команда отмены (undo).* Вполне логично использовать шаблон Command для реализации функции отмены. Достаточно расширить интерфейс Command методом undo, чтобы получить в реализующем классе практически весь набор функциональности, необходимой для отмены последней выполненной команды.

  Для того чтобы обеспечить отмену только последней команды, приложению достаточно постоянно сохранять ссылку на команду, выполненную последней. В таком случае, когда клиент выполняет операцию отмены, приложение вызывает метод undo лишь одного объекта, который и представляет собой последнюю команду.

  Однако во многих случаях возможность отмены одной только последней команды может показаться пользователю явно недостаточной. Для того чтобы обеспечить поддержку многоуровневой отмены, приложение должно отслеживать все выполнявшиеся команды, сохраняя их в хронологическом списке. Наличие такого списка также упрощает повторное выполнение одной и той же команды.

  Для того чтобы обеспечить корректную отмену команды, в шаблоне Command необходимо обеспечить наличие средств контроля за возможным нанесением ущерба. В частности, команда должна сохранять всю информацию, необходимую для восстановления измененного объекта, такую как сведения о получателе, передаваемые параметры, прежние значения атрибутов и т.п. Объект-получатель должен изменяться таким образом, чтобы команда могла восстановить исходные значения атрибутов.

  Необходимо напомнить, что команды могут выполняться по множеству раз в различных контекстах. Таким образом, прежде чем помещать команду в хронологический список, возможно, будет необходимо создать ее копию. Эта задача решается путем реализации шаблона Prototype (см. раздел Prototype на стр. 48).

  Создание копии объекта Command помогает избежать ошибок, которые могут возникать при постоянном выполнении последовательных операций отмены и повтора нескольких команд. Перемещение по хронологическому списку команд само по себе ничем не грозит, но если список реализован некорректно, это может повлечь за собой возникновение дополнительных ошибок. Для того чтобы избежать подобных ситуаций, в объекте команды нужно сохранять всю информацию, необходимую для корректной отмены операции. Если какие-то данные сохраняются внутри объекта-получателя, нужно прибегнуть к шаблону Memento (раздел Memento на стр. 105), который обеспечит сохранение состояния такого объекта. При использовании этого шаблона объект Memento может сообщить объекту Command всю информацию о предыдущем состоянии объекта-получателя. Таким образом, когда возникает необходимость в отмене команды, объект Command просто помещает объект Memento на место объекта-получателя.

- *Макрокоманда.* Объект`MacroCommand` — это коллекция объектов класса Command. Для создания макрокоманды можно воспользоваться шаблоном Composite. На рис. 2.5 представлена диаграмма классов для вариантов команды отмены и макрокоманды. (Более подробная информация о шаблоне Composite приведена на стр. 171).

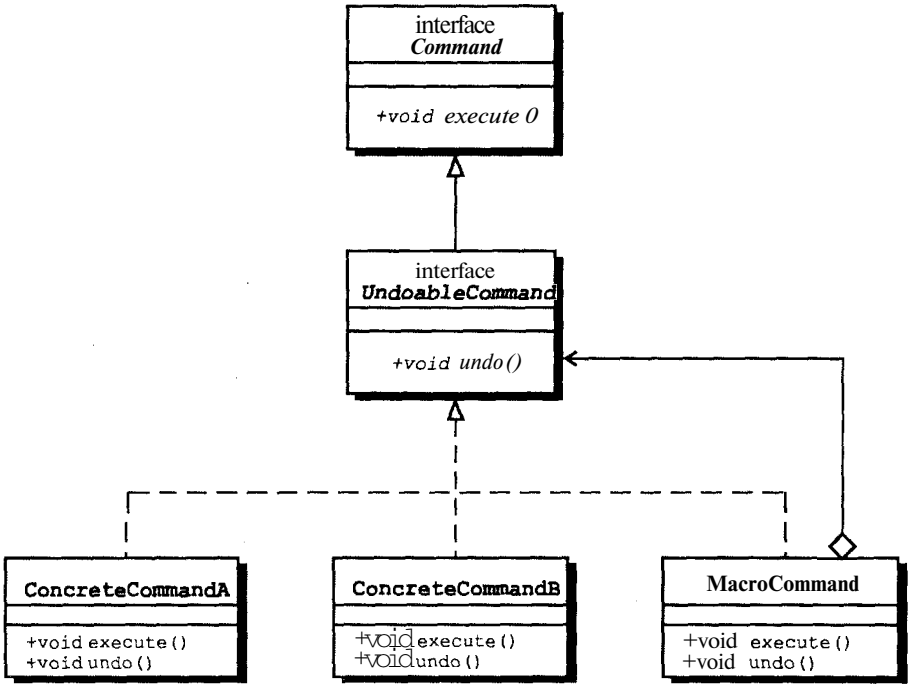


Рис. *2.5.* Диаграмма классов для команды отмены и макрокоманды

Коллекция MacroCommand содержит список подкоманд. Когда вызывается метод выполнения макрокоманды, коллекция переадресует вызов этого метода всем своим подкомандам.

Для того чтобы объект MacroCommand поддерживал операцию отмены, такую операцию должны поддерживать все хранящиеся в коллекции подкоманды. При вызове метода undo вызов должен передаваться дочерним объектам в порядке, обратном тому, в котором передается вызов метода `execute`.

Родственные шаблоны

К родственным относятся следующие шаблоны.

- Composite (стр. 171). Шаблон Composite используется для реализации макрокоманд.
- Memento (стр. 105). Сохраняет состояние получателя вместе с командой, обеспечивая тем самым возможность ее отмены.

- Prototype (стр. 48). Шаблон Prototype может использоваться для создания копии команды перед помещением ее в хронологический список.
- Singleton (стр. 54). В большинстве приложений хронологический список реализуется с помощью шаблона Singleton.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Command" на стр. 376 Приложения А.

Пользователям PIM-приложения может понадобиться обновление или же изменение хранящейся информации. В данном примере показано, как с помощью шаблона Command можно обеспечить выполнение функций обновления и отмены.

В данном случае поведение некоторой команды моделируется двумя интерфейсами. Базовая операция, выполняемая командой, определяется методом `execute` интерфейса Command (листинг 2.4). Интерфейс `UndoableCommand` (листинг 2.5) расширяет интерфейс Command методами отмены и повтора операций.

Листинг 2.4. `Command.java`

```
1. public interface Command{
2.   public void executed;
3. }
```

Листинг 2.5. `UndoableCommand.java`

```
1. public interface UndoableCommand extends Command{
2.   public void undo();
3.   public void redo();
4. }
```

Примером ситуации, в которой пользователю PIM-приложения может понадобиться команда отмены, является назначение места проведения запланированного события. Каждое событие (класс Appointment, представленный в листинге 2.6) содержит, помимо описания, списка приглашенных лиц и информации о времени начала и окончания, сведения о месте проведения.

Листинг 2.6. `Appointment.java`

```
1. import java.util.Date;
2. public class Appointment{
3.   private String reason;
4.   private Contact[] contacts;
5.   private Location location;
6.   private Date startDate;
```

```
7.  private Date endDate;
8.
9.  public Appointment(String reason, Contact[] contacts, Location location,
    Date startDate, Date endDate){
10.    this.reason - reason;
11.    this.contacts = contacts;
12.    this.location = =location;
13.    this.startDate = startDate;
14.    this.endDate = endDate;
15. }
16.
17. public String getReason(){ return reason; }
18. public Contact!] getContacts(){ return contacts; }
19. public Location getLocation(){ return location; )
20. public Date getStartDate() { return startDate; }
21. public Date getEndDate(){ return endDate; }
22.
23. public void setLocation(Location location){ this.location = location; }
24.
25. public String toString()(
26.    return "Appointment:" + "\n  Reason: " + reason +
27.    "\n    Location: " + location + "\n  Start; " +
28.    startDate + "\n  End: " + "\n";
29. )
30.}
```

Класс `ChangeLocationCommand` (листинг 2.7) реализует интерфейс `Undoable-Command` и снабжает приложение логикой, необходимой для изменения места проведения запланированного события.

Листинг 2.7. ChangeLocationCommand.java

```
1. public class ChangeLocationCommand implements UndoableCommand{
2.  private Appointment appointment;
3.  private Location oldLocation;
4.  private Location newLocation;
5.  private LocationEditor editor;
6.
7.  public Appointment getAppointment(){ return appointment; }
8.
9.  public void setAppointment(Appointment appointment){ this.appointment =
   appointment; }
10. public void setLocationEditor(LocationEditor locationEditor){ editor =
   locationEditor; }
11.
12. public void execute(){
13.    oldLocation = appointment.getLocation ();
14.    newLocation = editor.getNewLocation ();
15.    appointment.setLocation(newLocation);
16. }
17. public void undo(){
18.    appointment.setLocation(oldLocation);
19. }
20. public void redo(){
21.    appointment.setLocation(newLocation);
22. }
23.}
```

Класс ChangeLocationCommand дает пользователю возможность изменить место проведения события. Это достигается благодаря использованию приложением метода execute класса. С помощью метода undo класс обеспечивает сохранение предыдущего значения места проведения события и позволяет пользователю восстанавливать это значение, вызвав метод undo. Наконец, класс имеет метод redo, который дает пользователю возможность повторить назначение нового места проведения, если он того пожелает.

Interpreter

Свойства шаблона

Тип: поведенческий шаблон
Уровень: класс

Назначение

Определяет интерпретатор некоторого языка.

Представление

Как быстро вы можете собрать головоломку, представляющую собой разделенную на сотни фрагментов картинку? Можно допустить, что какая-то одаренная личность, лишь взглянув на 5000 фрагментов и проведя в уме некоторые расчеты, может с уверенностью сказать, как собрать головоломку.

Однако адепты другой школы применяют иной подход. Они рассортировывают все фрагменты, группируя их по принадлежности к отдельным частям картинки, а затем пытаются собрать сначала эти небольшие части. При этом фрагменты перебираются один за другим, пока не будет собрана небольшая часть картинки. Затем полученные части объединяются и весь процесс повторяется до тех пор, пока не будет собрана вся картинка (как всегда, в конце вы недосчитаетесь нескольких фрагментов, которые давным-давно потерялись).

При решении какой-то проблемы также часто прибегают к данному методу: разбивают сложную задачу на отдельные подзадачи меньшей сложности, затем эти подзадачи — на отдельные задания и т.д. Однако такой подход оказывается не слишком хорошим в тех случаях, когда подзадачи, задания и отдельные операции заданий взаимосвязаны между собой.

Лучшее решение состоит в создании простого языка, который описывает взаимосвязи между отдельными подзадачами. Затем остается промоделировать сложную проблему с помощью специального языка и найти описывающее ее выражение. При этом подходе необходимо иметь возможность значительно упростить процесс получения решения. Как и в случае с головоломкой, нужно разделить проблему на сравнительно меньшие части. После этого нужно сначала найти решения частных проблем, а затем объединить полученные решения для того, чтобы получить общее решение. При этом нужно надеяться, что по окончании работы все "фрагменты" окажутся на месте.

Область применения

Шаблон Interpreter применяется в следующих случаях.

- Необходимо интерпретировать простой язык.
- В терминах этого языка можно описать проблему с решениями, носящими итерационный характер.
- Отсутствуют жесткие требования к эффективности.

Описание

Шаблон Interpreter разделяет проблему на небольшие подзадачи, а затем представляет полученные фрагменты в виде выражения, записанного с помощью простого языка. Другая часть интерпретатора использует полученное выражение для поэтапной интерпретации и решения проблемы. Эта задача решается путем построения дерева абстрактного синтаксиса.

Широко известным примером данного подхода является *регулярное выражение* (*regular expression).* Регулярные выражения применяются для описания шаблонов, с помощью которых выполняется поиск и модификация строк, а используемый для этих целей язык очень скуп.

Рассмотрим некоторые термины на примере из области математики. Часто в приложениях приходится использовать формулы, например, знаменитую теорему Пифагора:

$$(A^2 + B^2) = C^2$$

Допустим, нам нужно сделать вычисления по простой математической формуле:

```
result = (a + b) / c
```

Таким образом, значение переменной `result` зависит от значений a, b и c.

Если значения этих переменных, соответственно, 4, 2 и 3, переменной `result` будет присвоено значение 2. Подумайте, как вы определили это? Во-первых, вы в уме "связали" a с 4, b с 2, а c — с 3. Затем вы сложили a с b, получив в результате значение 6, которое затем разделили на c (3).

Решение проблемы с помощью шаблона Interpreter выполняется примерно в такой же последовательности. Каждая из переменных (a, b и c) является *операндом* (operand), равно как и результат, полученный на очередном этапе вычислений.

Грамматические правила (например, + для обозначения сложения или / для обозначения деления) называются *операциями* (operations) или *операторами* (operators). Каждое грамматическое правило реализуется в виде отдельного класса, а каждое значение, находящееся справа от такого правила (значения также называются *операндами),* становится переменной экземпляра.

Реализация

Диаграмма классов шаблона Interpreter представлена на рис. 2.6.

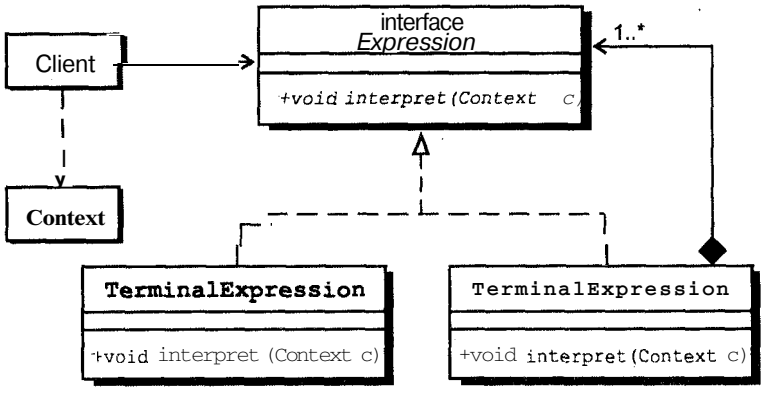


Рис. *2.6.* Диаграмма классов Interpreter

При реализации шаблона Interpreter обычно используются следующие классы.

- Expression. Интерфейс, через который клиент взаимодействует с выражением.
- TerminalExpression. Реализация интерфейса Expression, предназначенная для представления терминальных узлов грамматического и синтаксического деревьев.
- NonterminalExpression. Еще одна реализация интерфейса Expression, предназначенная для представления нетерминальных узлов грамматического и синтаксического деревьев. Объект этого класса хранит ссылку на следующий объект класса Expression, а также вызывает в случае необходимости метод interpret каждого из дочерних объектов.
- Context. Контейнер для информации, время от времени необходимой интерпретатору для выполнения определенных действий. Может служить в качестве коммуникационного канала для связи нескольких экземпляров класса Expression.
- Client. Создает или получает экземпляр абстрактного синтаксического дерева. Такое дерево образуется экземплярами классов TerminalExpression и NonterminalExpression для получения модели конкретного выражения. В случае необходимости клиент вызывает метод interpret с соответствующим контекстом.

Достоинства и недостатки

Можно отметить следующие достоинства и недостатки шаблона Interpreter.

- Интерпретатор очень простозаменить при изменении грамматики. Для того чтобы добавить правило, достаточно создать еще один класс, реализующий интерфейс Expression. Этот класс реализует новое правило в своем методе interpret.

Если нужно изменить правило, можно просто расширить старый класс и перекрыть в расширенном классе метод `interpret`.

- Шаблон Interpreter — не самое лучшее решение в тех случаях, когда грамматика языка сложна. Если в языке используется множество правил, шаблон Interpreter будет генерировать огромное количество классов, так как для каждого добавляемого правила шаблоном генерируется один или несколько классов. Чем обширнее грамматика языка, тем быстрее возрастает количество генерируемых шаблоном классов. Это может, в конечном итоге, привести к возникновению проблем на этапах отладки и эксплуатации.
- Объекты, представляющие выражения, можно использовать и для других целей. Например, можно добавить дополнительные методы в экземпляр класса Expression, чтобы расширить функциональность выражений. Чтобы улучшить гибкость приложения, используйте шаблон Visitor, который позволяет динамически изменять метод `interpret` (см. раздел "Visitor" на стр. 136).

Варианты

В оригинальном шаблоне, описанном в [GoF], вместо интерфейса задействован абстрактный класс. Как уже говорилось ранее, мы рекомендуем там, где это возможно, применять интерфейсы, а к абстрактным классам прибегать только в тех случаях, когда нужна частичная реализация.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Composite (стр. 171). Структура интерпретируемого выражения базируется на шаблоне Composite, использующем терминальные (узлы-листья) и нетерминальные выражения (узлы-ветви).
- Flyweight (стр. 196). Для уменьшения количества избыточных или подобных объектов можно при моделировании некоторых выражений воспользоваться шаблоном Flyweight.
- Iterator (стр. 87). Шаблон Iterator применяется для выполнения итераций при перемещении по абстрактному синтаксическому дереву и его узлам.
- Visitor (стр. 136). При использовании шаблона Visitor обеспечивается большая гибкость интерпретатора.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Interpreter" на стр. 383 Приложения ***А.***

Основа основ шаблона Interpreter — это иерархия классов Expression. Именно она определяет грамматику, которая используется для создания и разбора выраже-

ний. Интерфейс Expression — это не только база для всех выражений. Он также определяет метод `interpret`, который и выполняет анализ выражения.

В табл. 2.1 перечислены интерфейсы, обеспечивающие построение иерархии классов Expression, и соответствующая им информация.

Таблица 2.1. Назначение реализаций интерфейса Expression

Интерфейс	*Назначение*	*Листинг*
Expression	Общий интерфейс для всех выражений	2.8
ConstantExpression	Представляет константу	2.9
VariableExpression	Представляет переменную, полученную путем вызова метода некоторого класса	2.10
CompoundExpression	Пара выражений сравнения, при анализе которых получается логический результат	2.11
AndExpression	Логическое "И" для двух выражений	2.12
OrExpression	Логическое "ИЛИ" для двух выражений	2.13
ComparisonExpression	Пара выражений, при анализе которых получается логический результат	2.14
EqualsExpression	Выполняет метод **equals,** сравнивающий результаты двух выражений	2.15
ContainsExpression	Проверяет, содержит ли первое выражение, представленное в виде строки, вторую строку	2.16

Листинг 2.8. Expression.java

```
1. public interface Expression{
2.   void interpret(Context c);
3. }
```

Листинг 2.9. ConstantExpression.java

```
1. import java.lang.reflect.Method;
2. import java.lang.reflect.InvocationTargetException;
3. public class ConstantExpression implements Expression{
4.   private Object value;
5.
6.   public ConstantExpression(Object newValue){
7.     value = newValue;
8.   }
9.
10.  public void interpret(Context c) {
11.    c.addVariable(this, value);
12.  }
13.}
```

Листинг 2.10. VariableExpression.java

```
1. import java.lang.reflect.Method;
2. import java.lang.reflect.InvocationTargetException;
3. public class VariableExpression implements Expression{
4.  private Object lookup;
5.  private String methodName;
6.
7.  public VariableExpression(Object newLookup, String newMethodName){
8.     lookup = newLookup;
9.    methodName = newMethodName;
10. }
11.
12. public void interpret(Context c){
13.    try{
14.       Object source = c.get(lookup);
15.       if (source != null){
16.           Method method = source.getClass().getMethod(methodName, null);
17.           Object result = method.invoke(source, null);
18.           c.addVariable(this, result);
19.       }
20.    }
21.    catch (NoSuchMethotException exc) { }
22.    catch (IllegalAccessException exc){ }
23.    catch (InvocationTargetException exc)( )
24. }
25.}
```

Листинг 2.11. CompoundExpression.java

```
1. public abstract class CompoundExpression implements Expression{
2.  protected ComparisonExpression expressionA;
3.  protected ComparisonExpression expressionB;
4.
5.  public CompoundExpression(ComparisonExpression expressionA,
   ComparisonExpression expressionB){
6.     this.expressionA = expressionA;
7.     this.expressionB = expressionB;
8.  }
9. }
```

Листинг 2.12. AndExpression.java

```
1. public class AndExpression extends CompoundExpression{
2.  public AndExpression(ComparisonExpression expressionA, ComparisonExpression
   expressionB){
3.     super(expressionA, expressionB);
4. }
5.  public void interpret(Context c){
6.     expressionA.interpret(c) ;
7.     expressionB.interpret(c) ;
8.     Boolean result = new
   Boolean(((Boolean)c.get(expressionA)).booleanValue() &&
    ((Boolean)c.get(expressionB)).booleanValue());
9.     c.addVariable(this, result);
10. }
11.}
```

Листинг 2.13. OrExpression.java

```
1. public class OrExpression extends CompoundExpression{
2.   public OrExpression(ComparisonExpression expressionA, ComparisonExpression
   expressionB){
3.     super(expressionA, expressionB);
4.   }
5.   public void interpret(Context c) (
6.      expressionA.interpret(c);
7.      expressionB.interpret(c);
8.      Boolean result = new-
   Boolean ( ((Boolean)c.get(expressionA)).booleanValue() ||
    ((Boolean)c.get(expressionB)).booleanValue());
9.      c.addVariable(this, result);
10. }
11.}
```

Листинг 2.14. **ComparisonExpression.java**

```
1. public abstract class ComparisonExpression implements Expression)
2.   protected Expression expressionA;
3.   protected Expression expressionB;
4.
5.   public ComparisonExpression(Expression expressionA, Expression
   expressionB){
6.     this .expressionA = expressionA;
7.      this.expressionB = expressionB;
8.   }
9. }
```

ЛИСТИНГ 2.15. **EqualsExpression.java**

```
1. public class EqualsExpression extends ComparisonExpression{
2.   public EqualsExpression(Expression expressionA, Expression expressionB{
3.     super expressionA, expressionB) ;
4.   }
5.
6.   pub_ic void interpret (Context c) (
7.     expressionA.interpret(c) ;
8.     expressionB.interpret(c) ;
9.     Boolean result = new Boolean
    (c.get(expressionA).equals(c.get(expressionB))) ;
10.    c.addVariable(this, result) ;
11. }
12.}
```

Листинг **2.16. ContainsExpression.java**

```
1. public class ContainsExpression extends ComparisonExpression{
2.   public ContainsExpression(Expression expressionA, Expression expressionB){
3.     super (expressionA, expressionB) ;
4. }
5.
6.   public void interpret (Context c){
```

```
7.      expressionA.interpret(c) ;
8.      expressionB.interpret(c) ;
9.      Object exprAResult = c.get(expressionA) ;
10.     Object exprBResult = c.get expressionB) ;
11.     if ((exprAResult instanceof String) && (exprBResult instanceof String)){
12.       if (((String)exprAResult) .indexOf((String)exprBResult) != -1){
13.           c.addVariable(this, Boolean.TRUE) ;
14.           return;
15.       )
16.     }
17.     c.addVariable(this, Boolean.FALSE) ;
18.     return;
19.  }
20.}
```

Класс Context (листинг 2.17) предоставляет память, необходимую для анализа выражений. Этот класс по сути является оболочкой (wrapper) класса HashMap. В данном примере объекты класса Expression предоставляют классу HashMap ключи, по которым в хеш-памяти сохраняются результаты вызова метода interpret.

Листинг 2.17. Context.java

```
1. import java.util.HashMap;
2. public class Context{
3.   private HashMap map = new HashMap ();
4.
5.   public Object get (Object name){
6.      return map.get(name);
7.   }
8.
9.   public void addVariable(Object name, Object value){
10.     map.put(name, value);
11.  }
12.)
```

Имея такой набор базовых выражений, можно выполнять довольно сложные операции. Рассмотрим объект класса ContactList (листинг 2.18), в котором содержатся сведения о контактных лицах. У класса имеется метод getContactsMatching-Expression, который для каждого объекта Contact вычисляет результат, представленный объектом класса Expression, и возвращает объект класса ArrayList.

Листинг 2.18. ContactList.java

```
1. import.java.io.Serializable;
2. import.java.util.ArrayList;
3. import.java.util.Iterator;
4. public class ContactList implements Serializable{
5.   private ArrayList contacts = new ArrayList();
6.
7.   public ArrayList getContacts(){ return contacts; }
8.   public Contact [] get ContactsAsArray(){ return (Contact []
   (contacts.toArray(new Contact [1])); }
9.
10.  public ArrayList getContactsMatchingExpression(Expression expr, Context
   ctx, Object key){
11.     ArrayList results = new ArrayList();
```

```
12.    Iterator elements = contacts.iterator();
13.    while (elements.hasNext(){
14.     Object currentElement = elements .next 0;
15.     ctx.addVariable(key, currentElement);
16.      expr.interpret(ctx);
17.     Object interpretResult = ctx.get(expr);
18.     if ((interpretResult != null)
   &&(interpretResult.equals(Boolean.TRUE))){
19.           results.add(currentElement);
20.      }
21.    }
22.   return results;
23. }
24.
25. public void setContacts(ArrayList newContacts){ contacts = newContacts; )
26.
27. public void addContact(Contact element){
28.    if (!contacts.contains(element)){
29.      contacts.add(element);
30.    }
31. )
32. public void removeContact(Contact element){
33.   contacts.remove(element) ;
34. }
35.
36. public String toString(){
37.    return contacts.toString() ;
38. }
39.}
```

Используя иерархию объектов класса Expression и объект класса ContactList, можно выполнять запросы на поиск объектов класса Contact к объекту класса ContactList так, словно последний является базой данных. Например, можно выполнить поиск всех контактных лиц, в названии должности которых используется слово "Java". Для этого нужно выполнить следующие операции.

1. Создать объект класса ConstantExpression и инициализировать его строкой "Java".
2. Создать объект класса VariableExpression и инициализировать его ссылкой на объект класса Contact и строкой "getTitle".
3. Создать объект класса ContainsExpression, передав ему в качестве первого параметра ссылку на объект класса VariableExpression, а в качестве второго — ссылку на объект класса ConstantExpression.
4. Передать ссылку на объект класса ContainsExpression методу getContacts-MatchingExpression объекта класса ContactList.

Iterator

Также известен как Cursor

Свойства шаблона

Тип: поведенческий шаблон
Уровень: компонент

Назначение

Предоставляет единый метод последовательного доступа к элементам коллекции, не зависящий от самой коллекции и никак с ней не связанный.

Представление

В связи с тем, что PIM-приложением обрабатывается много различных структурированных данных, в нем для хранения этих данных интенсивно используются коллекции. Адреса, контакты, проекты, события, заметки, списки неотложных дел — все эти данные должны сохраняться в виде групп взаимосвязанных объектов.

Чтобы обеспечить соблюдение всех требований к хранению подобных данных, разрабатываются специальные классы, предназначенные для хранения каждой группы элементов одного типа. Такие классы, учитывающие особенности каждой группы хранящихся в ней элементов, и называются коллекциями.

Однако если возникает необходимость последовательного перебора элементов коллекции, мы можем столкнуться с небольшой проблемой. Если классы коллекций проектировались таким образом, чтобы как можно лучше соответствовать хранящимся в них элементам, у нас нет гарантий, что все элементы всех коллекций будут извлекаться из них одним и тем же способом. Например, запланированные события могут быть организованы в подгруппы по определенной дате, сведения же о контактных лицах скорее всего сохраняются в алфавитном порядке, а заметки — в порядке их создания.

Это означает, что для каждой коллекции придется разрабатывать свой механизм извлечения элементов, дублируя его в разных местах системы. Теоретически это может привести к получению очень сложного и неудобного в эксплуатации приложения. Более того, нужно очень хорошо разбираться во всех подробностях каждой коллекции, используемой для хранения прикладных объектов PIM-приложения.

Все эти проблемы решаются путем использования шаблона Iterator, который определяет универсальный интерфейс, предназначенный для перемещения по элементам любой коллекции. Когда в системе используются итераторы, разработчик может применять одинаковые вызовы методов как для перемещения по списку контактов, так и для распечатки перечня неотложных дел.

Область применения

Шаблон Iterator рекомендуется использовать в следующих случаях.

- Необходимо обеспечить единый механизм перебора элементов коллекций, не зависящий от способа реализации ни одной из этих коллекций.
- Необходимо обеспечить множественный доступ к коллекции, позволяющий нескольким клиентам одновременно работать с одной и той же коллекцией.

Описание

Шаблон Iterator позволяет прежде всего стандартизировать и упростить исходный код приложения, с помощью которого обеспечивается доступ к коллекциям. При создании классов коллекций основное внимание уделяется их средствам сохранения ин-

формации, а не ее извлечения. Достоинством шаблона Iterator является то, что он обеспечивает единый подход к организации перемещения по самым разным коллекциям, не зависящий от их внутренней структуры.

В языке программирования Java шаблон Iterator обычно определяется как интерфейс, на основе которого создаются одна или несколько реализаций, на которые возлагается задача взаимодействия с нижележащими механизмами хранения информации. Шаблон Iterator, описанный в [GoF], обеспечивает выполнение следующих фундаментальных операций.

- `First`
- `Next`
- `IsDone`
- `CurrentItem`

Эти операции представляют собой основные функции, которые итератор должен выполнять в любой ситуации. Иными словами, итератор должен обеспечивать поддержку следующих возможностей.

- *Перемещение.* Обеспечение последовательного перебора элементов коллекции в обоих направлениях.
- *Извлечение.* Обеспечение возможности получения ссылки на текущий элемент коллекции.
- *Проверка.* Определение, существуют ли еще элементы в коллекции, путем проверки значения текущей позиции итератора.

Кроме того, итераторы могут выполнять и другие операции. Некоторые, например, содержат методы, позволяющие перейти к первому или последнему элементу коллекции.

Реализация

Диаграмма классов шаблона Iterator представлена на рис. 2.7.
При реализации шаблона Iterator обычно используются следующие классы.

- `Iterator`. Интерфейс, определяющий стандартные методы последовательного перемещения по коллекции. Такой интерфейс должен содержать, как минимум, методы перемещения, извлечения и проверки (`first`, `next`, `hasMoreElements` и `getCurrentItem`).
- `ConcreteIterator`. Класс, реализующий интерфейс `Iterator`. Такие классы содержат ссылки на нижележащие коллекции. Обычно экземпляры класса `ConcreteIterator` создаются с помощью класса ConcreteAggregate. Благодаря его тесной связи с классом ConcreteAggregate, класс `ConcreteIterator` часто создается в виде внутреннего класса ConcreteAggregate.
- `Aggregate`. Интерфейс, определяющий метод, на который возлагается задача генерации экземпляров класса `Iterator`.

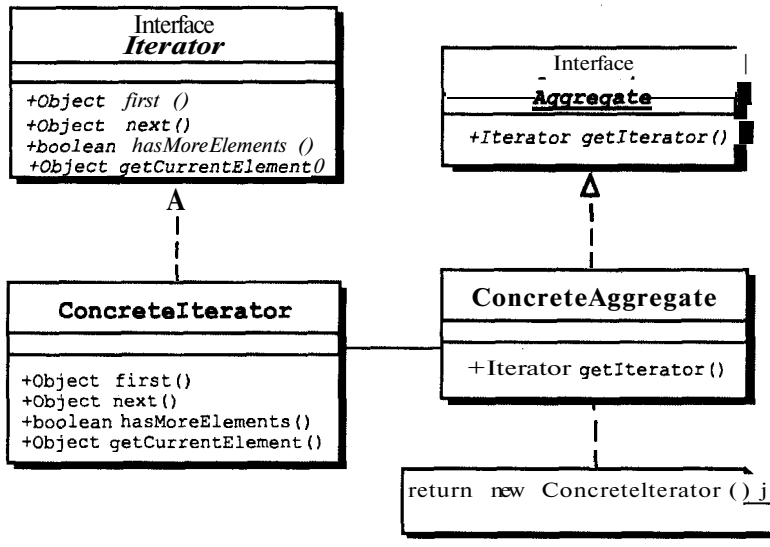


Рис. 2.7. Диаграмма классов шаблона Iterator

- ConcreteAggregate. Класс, реализующий интерфейс Aggregate и создающий по мере необходимости экземпляры класса ConcreteIterator. Помимо этой задачи, класс ConcreteAggregate выполняет и свою основную функцию — представляет в системе коллекцию объектов. Класс ConcreteAggregate создает экземпляр класса ConcreteIterator.

Достоинства и недостатки

Многие из достоинств шаблона Iterator основываются на преимуществах, которые предоставляет единый интерфейс перемещения по коллекции. Это в значительной степени упрощает работу с коллекциями, а также позволяет широко использовать при разработке коллекций такое свойство, как полиморфизм. Например, для того чтобы распечатать список элементов, хранящийся в любой коллекции, нужно получить ссылку на соответствующий объект класса Iterator, а затем вызвать метод toString любого объекта независимо от того, как устроена коллекция, в которой этот объект хранится.

Кроме того, итераторы позволяют нескольким клиентам работать одновременно с одной и той же коллекцией. Можно представить итератор, как указатель на очередной объект в коллекции. При каждом вызове метода класса Aggregate мы получаем указатель на следующий объект коллекции.

Недостатком итераторов является то, что их применение создает иллюзию упорядоченности неупорядоченных структур. Например, множество не поддерживает возможности упорядочивания, поэтому его итератор будет выдавать элементы в случайной последовательности, которая может время от времени меняться. Если разработчик не знает об этой особенности, он может написать код, основанный на предположении о том, что нижележащая структура итератора упорядочена. Впоследствии это может привести к возникновению тех или иных проблем.

Варианты

Шаблон Iterator может реализовываться множеством различных способов.

- Класс `ConcreteIterator` может быть как внутренним, так и внешним. Внешние итераторы предоставляют клиентам специальные методы, позволяющие перемещаться по коллекции, тогда как внутренние итераторы сами перебирают элементы коллекции, выполняя запросы клиентов. Таким образом, внешний итератор обладает большей гибкостью, но требует при этом и большего объема работы по программированию взаимодействия клиента с итератором.
- Класс `Concretelterator` может быть как динамическим, так и статическим. Динамический итератор ссылается непосредственно на объекты, содержащиеся в нижележащей коллекции, поэтому выдаваемые им результаты всегда соответствуют реальному состоянию коллекции. Статический же итератор в момент своего создания получает копию коллекции, поэтому при обращении клиента выдает ссылки на объекты, хранящиеся в копии, а не в оригинальной коллекции.
- Для упрощения задачи перемещения по сложным структурам, таким как деревья, могут создаваться пустые итераторы. Используя итераторы, которые представляют "оконечный узел", можно написать простой рекурсивный код, обеспечивающий обход всех узлов дерева.
- Итераторы могут поддерживать самые разные методы перемещения по коллекциям. Это особенно важно при их использовании в сложных структурах, таких как шаблон Composite, когда наличие разных методов перемещения может играть весьма важную положительную роль.
- Что касается структуры, класс `Concretelterator` может выполняться как внутренний класс класса ConcreteAggregate или в виде отдельного класса. При этом класс `Concretelterator` может содержать весь код, необходимый для перемещения по коллекции, или же лишь отображать текущую позицию в коллекции.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Factory Method (стр. 42). Классы коллекций часто содержат специальный метод, предназначенный для генерации итератора.
- Visitor (стр. 136). При использовании шаблона Visitor для группы объектов часто для перемещения по элементам группы используют итератор.
- Value List Handler [CJ2EEP]. Шаблон Value List Handler базируется на шаблоне Iterator, что позволяет клиентам перемещаться по коллекциям.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Iterator" на стр. 391 Приложения А.

Данный пример построен на использовании коллекций из стандартной библиотеки языка Java. Для работы с коллекциями интерфейс `java.util.Iterator` определяет основные методы, предназначенные для обеспечения навигации, — hasNext и next. Необходимо отметить, что интерфейс `Iterator` обеспечивает возможность выполнения только одного прохода по коллекции. Это означает, что если приложению понадобиться вернуться к одному из первых элементов коллекции, приложению придется создать новый экземпляр класса `Iterator`.

Интерфейс `Iterating` (листинг 2.19) содержит определение лишь одного метода, названного `getIterator`. Этот интерфейс используется для идентификации любого класса, входящего в состав PIM-приложения, как такого, который может генерировать итератор коллекции.

Листинг 2.19. Iterating.java

```
1. import java.util.Iterator;
2. import java.io.Serializable;
3. public interface Iterating extends Serializable{
4.   public Iterator getIterator();
5. i
```

Интерфейсы ToDoList (листинг 2.20) и `ToDoListCollection` (листинг 2.21), расширяющие интерфейс `Iterating`, используются в данном примере для определения двух коллекций. Интерфейс ToDoList предназначен для представления классов, обеспечивающих хранение списка неотложных дел, а интерфейс `ToDoListCollection` — для представления коллекций классов ToDoList, хранимых PIM-приложением.

Листинг 2.20. ToDoList.java

```
1. public interface ToDoList extends Iterating{
2.   public void add(String item);
3.  public void add(String item, int position);
4.   public void remove(String item);
5.  public int getNumberOfItems ();
6.   public String getListName();
7.   public void setListName(String newListName);
8. }
```

Листинг 2.21. ToDoListCollection.java

```
1. public interface ToDoListCollection extends Iterating{
2.   public void add(ToDoList list);
3.   public void remove(ToDoList list);
4.   public int getNumberOfIterns ();
5. }
```

Приведенные выше интерфейсы реализуются, соответственно, классами `ToDoListImpl` и `ToDoListCollectionImpl`. Класс `ToDoListImpl` (листинг 2.22) обеспечивает хранение своих элементов путем использования класса ArrayList, который позволяет упорядочивать элементы по абсолютным значениям, а также допускает дублирование элементов. Класс `ToDoListCollectionImpl` (листинг 2.23) использует

класс HashTable, который не поддерживает упорядочения и сохраняет все элементы в виде пар "ключ — значение". Хотя, как видно, поведение коллекций весьма различно, у обоих имеются итераторы, обеспечивающие навигацию по коллекциям.

Листинг 2.22. ToDoListImpl.java

```
1. import java.util.Iterator;
2. import java.util.ArrayList;
3. public class ToDoListImpl implements ToDoList{
4.   private String listName;
5.   private ArrayList items = new ArrayList ();
6.
7.   public void add(String item){
8.      if (!items.contains(item)){
9.        items.add(item) ;
10.     }
11. }
12.  public void add(String item, int position){
13.     if (!items.contains(item)){
14.       items.add(position, item);
15.     }
16. }
17.  public void remove(String item){
18.     if (items.contains(item)){
19.       items.remove(items.indexOf(item)) ;
20.     }
21. )
22.
23.  public int getNumberOfItems(){ return items.size();}
24.  public Iterator getIterator () { return items.iterator (); }
25.  public String getListName(){ return listName;}
26.  public void setListName(String newListName){ listName = newListName; }
27.
28.  public String toString(){ return listName;}
29.}
```

Листинг 2.23. ToDoListCollectionImpl.java

```
1. import java.util.Iterator;
2. import java.util.HashMap;
3. public class ToDoListCollectionImpl implements ToDoListCollection{
4.   private HashMap lists = new HashMap ();
5.
6.   public void add(ToDoList list){
7.      if (!lists.containsKey(list.getListName())){
8.        lists.put(list.getListName(), list);
9.      }
10. }
11.  public void remove(ToDoList list){
12.     if (lists.containsKey(list.getListName())){
13.      lists.remove(list.getListName() ) ;
14.     }
15. }
16.  public int getNumberOfItems() { return lists.size(); }
17.  public Iterator getIterator(){ return lists.values().iterator(); }
18.  public String toString(){ return getClass().toString(); }
19.}
```

Так как оба класса предоставляют доступ к итератору, достаточно просто написать код, который обеспечивает перемещение по элементам соответствующих коллекций. Так, на примере класса ListPrinter (листинг 2.24) показано, как использовать итератор для вывода на печать содержимого коллекций, представленного в строковом виде. Класс имеет три метода: printToDoList, printToDoListCollection и printIteratingElement. В каждом из методов последовательный перебор всех элементов коллекций организован с использованием очень простого цикла while.

Листинг 2.24. ListPrinter.java

```
1. import java.util.Iterator;
2. import java.io.PrintStream;
3. public class ListPrinter{
4.   public static void printToDoList(ToDoList list, PrintStream output){
5.     Iterator elements = list.getIterator();
6.    output.println(" List - "+ list + " : " ) ;
7.     while (elements.hasNext() ){
8.       output.println("\t" +elements.next());
9.     }
10. }
11.
12.  public static void printToDoListCollection(ToDoListCollection lotsOfLists,
   PrintStream output){
13.     Iterator elements = lotsOfLists.getIterator();
14.    output.println("\"To Do\" ListCollection:");
15.     while (elements.hasNext()){
16.      printToDoList((ToDoList)elements.next(), output);
17.    }
18. }
19.
20.  public static void printIteratingElement(Iterating element, PrintStream
   output){
21.    output.println("Printing the element " + element);
22.     Iterator elements = element.getIterator();
23.     while (elements.hasNext()){
24.      Object currentElement = elements.next();
25.      if (currentElement instanceof Iterating){
26.          printIteratingElement((Iterating)currentElement, output);
27.           output.println();
28.      }
29.      else{
30.           output.println(currentElement);
31.      }
32.    }
33. }
34.}
```

Мощь комбинации шаблона Iterator и полиморфизма лучше всего видна на примере метода printIteratingElement. В нем, в частности, показано, как распечатать в строковом представлении данные любого класса, реализующего интерфейс Iterating. При этом методу ничего не нужно знать о структуре нижележащей коллекции за исключением того, что она может генерировать итераторы.

Mediator

Свойства шаблона

Тип: поведенческий шаблон
Уровень: компонент

Назначение

Предназначен для упрощения взаимодействия объектов системы путем создания специального объекта, который управляет распределением сообщений между остальными объектами.

Представление

PIM-приложение очень выиграет от реализации в нем возможности совместного доступа к информации. Это позволит, например, одному пользователю запланировать какое-то мероприятие, а другим пользователям, которые должны быть участниками этого мероприятия, сразу же узнать об этом, увидев изменения в своем рабочем расписании. Если обеспечивается совместный доступ к данным, то все участники встречи будут автоматически видеть изменения в своих планах, вносимых любым из участников.

Однако как управлять процессом одновременного планирования событий разными пользователями, учитывая тот факт, что каждый из этих пользователей работает с собственной копией PIM-приложения? Можно, например, предоставить каждой копии приложения доступ к копии объекта Appointment, обеспечив ей в свою очередь доступ к локальным данным. Однако такой подход порождает новую проблему — как теперь обеспечить целостность информации одновременно для всех пользователей? Например, если пользователь запланировал встречу, а затем изменил дату ее проведения, как остальные участники смогут об этом узнать?

Конечно, можно возложить задачу обновления информации на экземпляр приложения, с которым работает пользователь. Однако для того, чтобы каждый участник встречи мог обновлять планы, затрагивающие других участников, нужно чтобы каждый пользовательский экземпляр PIM-приложения отслеживал все изменения, вносимые остальными экземплярами PIM-приложений других пользователей. Понятно, что управление взаимодействием множества клиентов — задача не из легких. В лучшем случае такое решение окажется малоэффективным и требовательным относительно сетевых ресурсов. В худшем же случае планирование собрания превратится в кошмар.

Учитывая сложность системы, лучше всего делегировать задачу отправки и получения определенных запросов некоторому центральному объекту, принимающему решения о том, какой именно метод необходимо вызвать для обработки того или иного запроса. Для решения этой задачи и предназначен шаблон Mediator. Разработчик, вместо того, чтобы возлагать ответственность за обновление информации на объект Appointment, создает объект-посредник класса `AppointmentMediator`. Теперь при каждом изменении в данных объекта Appointment вызывается опреде-

ленный метод объекта-посредника, который в свою очередь может вызвать соответствующий метод объекта, который должен обновить свою информацию. В зависимости от принятого решения объект AppointmentMediator рассылает либо исходное сообщение, либо его пересмотренную версию (например, с изменением времени проведения), либо сообщение об отмене встречи.

Областьприменения

Шаблон Mediator рекомендуется использовать в следующих случаях.

- Имеется сложный набор правил, определяющих взаимодействие объектов в системе (часто сложность продиктована самой бизнес-моделью).
- Необходимо обеспечить простоту выполнения и управляемость объектами.
- Необходимо обеспечить возможность повторного использования объектов, не привязываясь к конкретной бизнес-модели системы.

Описание

По мере того, как усложняется механизм взаимодействия объектов системы, многократно усложняется задача управления этим взаимодействием. Создание обработчика событий для простого объекта электронной таблицы может сводится к разработке одного лишь компонента — сетки. Однако если в пользовательский интерфейс, кроме сетки, добавляются диаграмма и поля базы данных, задача обработки событий приложения значительно усложняется. Это усложнение продиктовано, прежде всего, тем, что внесение изменений в один компонент может повлечь за собой изменения в других компонентах или даже всей системы в целом.

Шаблон Mediator позволяет решить эту проблему, так как на его основе определяется один класс, на который возлагается задача по обеспечению взаимодействия всех или некоторых объектов приложения. Данное решение позволяет значительно упростить остальные классы системы, так как им уже не нужно самим управлять взаимодействием, а это в свою очередь значительно упрощает жизнь разработчику. Объект-посредник, т.е. объект по централизованному управлению взаимодействием других объектов, играет в системе роль маршрутизатора, в котором централизована логика по отправке и получению сообщений. В такой системе компоненты отправляют сообщения не другим объектам, а объекту-посреднику. Ему же отправляются все сообщения о внесении изменений.

В тех случаях, когда нужно работать с компонентами пользовательского интерфейса, как с единым целым, можно реализовать шаблон Mediator. Основным фактором, влияющим на принятие решения о его применении, является уровень сложности модели пользовательского интерфейса. Среди других возможных сценариев реализации шаблона Mediator можно выделить следующие два.

- Независимая часть бизнес-модели, например, продукт, состоящий из нескольких компонентов.
- Модель, представляющая законченный процесс, такой как последовательная обработка заказа бухгалтерией, производственным отделом и отделом доставки.

В качестве примера "реализации" шаблона Mediator в реальном мире можно привести пульт для проведения телеконференций. В нем "прошита" определенная логика (вне всякого сомнения), обеспечивающая пересылку сообщений между участниками телеконференции. Участники отправляют свои сообщения (т.е. попросту говорят по телефону), а пульт управления перенаправляет их адресатам. При этом одни сообщения уходя в Бирму или Антверпен, тогда как сообщения, представляющие собой вызов руководителя, отправляются тому, кто ведет телеконференцию.

Реализация

Диаграмма классов шаблона Mediator представлена на рис. 2.8.

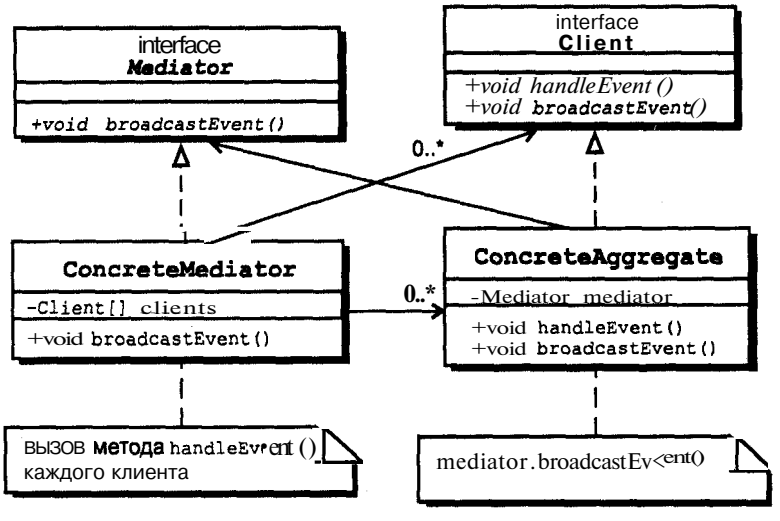


Рис. *2.8.* Диаграмма классов шаблона Mediator

При реализации шаблона Mediator обычно используются следующие классы.

- Mediator. Интерфейс, определяющий методы, с помощью которых клиенты могут обращаться к объекту-посреднику.
- ConcreteMediator. Класс, реализующий интерфейс Mediator. Именно этот класс и играет роль посредника, взаимодействующего с несколькими клиентскими классами. Он содержит определяемую приложением информацию о процессах. Кроме того, ConcreteMediator может содержать среди своих данных "жесткие" ссылки на клиентов. Основываясь на получаемой информации, посредник либо вызывает те или иные методы клиента, либо задействует некий общий метод, информирующий клиентов об изменении, либо выполняет обе операции.
- Client. Интерфейс, определяющий общие методы, посредством которых посредник может информировать экземпляры классов клиентов.

- ConcreteClient. Класс, реализующий интерфейс Client и обеспечивающий реализацию каждого метода клиента. Класс ConcreteClient может среди прочих данных сохранять ссылку на экземпляр класса Mediator, чтобы через последний информировать других клиентов о тех или иных изменениях в своих данных.

Достоинства и недостатки

Шаблон Mediator обладает тремя следующими достоинствами.

- Позволяет упростить отдельные компоненты, так как избавляет их от необходимости организации прямого обмена сообщениями. Компоненты становятся более универсальными, поскольку им не нужно обеспечивать взаимодействие с другими компонентами. Таким образом, вся специфическая для приложения логика сосредотачивается в объекте-посреднике.
- Обеспечивает упрощение поддержки всей стратегии взаимодействия объектов, так как вся ответственность за него возлагается на объект-посредник.
- Очевидно, что сам по себе объект-посредник предназначен для конкретного приложения и его трудно применить в каком-то другом приложении. В этом нет ничего удивительного, так как посредник создается именно для того, чтобы инкапсулировать в себе все свойственное приложению поведение, чтобы обеспечить универсальность другим классам системы. Централизация свойственного приложению поведения в объекте-посреднике значительно облегчает сопровождение приложения. Кроме того, разработчик может повторно использовать остальные классы в других приложениях: все, что для этого нужно, — это лишь переписать класс посредника, приспособив его к нуждам нового приложения.

Тестирование и отладка сложной реализации шаблона Mediator может оказаться нелегким делом — разработчику придется внимательно протестировать и компонент, образующий объект-посредник, и связанные с ним объекты.

По мере возрастания количества объектов в системе код объекта-посредника может стать трудноуправляемым. Одним из возможных решений в такой ситуации может стать преобразование посредника в сложный объект, состоящий из множества специализированных независимых посредников. (Более подробная информация о шаблоне Composite приведена в разделе "Composite" на стр. 171.) При таком подходе объект-посредник представляет собой объект централизованного управления, связанный со множеством классов отдельных служб, каждый из которых обеспечивает определенный набор функциональности.

Варианты

Шаблон Mediator можно реализовывать в самых разных вариантах.

- *Однонаправленные коммуникации.* Некоторые реализации позволяют клиентам системы либо только отправлять сообщения, либо только получать их.

- *Распараллеливание.* Подобно многим другим поведенческим шаблонам, шаблон Mediator является одним из кандидатов на многопоточную обработку. Если объект-посредник поддерживает многопоточность, в нем может реализовываться очередь сообщений, а также присутствовать поддержка других возможностей, например, управление по приоритетам.
- *Настраиваемые роли.* В данном варианте клиенты определяют роли (которые в общем случае могут меняться в ходе работы системы), выдвигающие определенные требования к сообщениям. Хотя этот вариант весьма трудно реализовать, все же определение объектов в терминах ролей может позволить получить более универсальный класс **объекта-посредника**, что облегчает его повторное использование в других системах.
- *Запросы клиентов.* Посредник может сохранять подробные сообщения, отправляя клиентам лишь краткие извещения. Затем клиенты сами запрашивают подробную информацию, если она им нужна.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Observer (стр. 111). Этот шаблон часто используется для управления взаимодействием клиента и посредника локальной системы. Так как в системе часто реализуется один посредник, его нередко выполняют в соответствии с шаблоном Singleton или в виде ресурсов уровня класса (путем объявления методов класса посредника статическими).
- HOPP (стр. 202). Шаблон Mediator, **реализованный** в сети, для обеспечения коммуникации может пользоваться шаблоном Half-Object Plus Protocol **(HOPP)**.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Mediator" на стр. 396 ***Приложения А.***

В данном примере шаблон Mediator используется для управления взаимодействием панелей графического пользовательского интерфейса. В базовом варианте данного GUI имеются панели для выбора контактного лица из списка, для редактирования и для отображения текущего состояния контакта. Посредник взаимодействует с каждой панелью, вызывая методы, которые предназначены для обновления элементов GUI в соответствии со вносимыми изменениями.

Класс MediatorGui (листинг 2.25) создает в приложении главное окно и три панели. Кроме того, он создает объект-посредник и связывает его с тремя дочерними панелями.

Листинг 2.25. MediatorGui.java

```
1. import java.awt.Container;
2. import java.awt.event.WindowEvent;
3. import java.awt.event.WindowAdapter;
4. import javax.swing.BoxLayout;
5. import javax.swing.JButton;
6. import javax.swing.JFrame;
7. import javax.swing.JPanel ;
8. public class MediatorGui{
9.   private ContactMediator mediator;
10.
11.  public void setContactMediator(ContactMediator newMediator){ mediator -
   newMediator; }
12.
13.  public void createGui(){
14.    JFrame mainFrame = new JFrame("Mediator example");
15.     Container content = mainFrame.getContentPane();
16.     content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
17.     ContactSelectorPanel select = new ContactSelectorPanel(mediator);
18.     ContactDisplayPanel display = new ContactDisplayPanel(mediator);
19.     ContactEditorPanel edit - new ContactEditorPanel(mediator);
20.     content.add(select);
21.     content.add(display);
22.     content.add(edit);
23.    mediator.setContactSelectorPanel(select) ;
24.     mediator.setContactDisplayPanel(display);
25.    mediator.setContactEditorPanel(edit) ;
26.     mainFrame.addWindowListener(new WindowCloseManager());
27.     mainFrame.pack();
28.    mainFrame.setVisible(true) ;
29.  }
30.  private class WindowCloseManager extends WindowAdapter{
31.     public void windowClosing(WindowEvent evt){
32.        System.exit(0);
33.     )
34.  }
35.}
```

Самой простой панелью GUI является панель, представленная объектом класса ContactDisplayPanel (**листинг 2.26**). В этом классе имеется метод, названный contactChanged, который обновляет представление объекта на экране в соответствии с новыми значениями параметра Contact.

Листинг 2.26. ContactDisplayPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JScrollPane;
4. import javax.swing.JTextArea:
5. public class ContactDisplayPanel extends JPanel{
6.   private ContactMediator mediator;
7.   private JTextArea displayRegion;
8.
9.   public ContactDisplayPanel(){
10.     createGui();
11.  }
12.   public ContactDisplayPanel(ContactMediator newMediator){
```

```
13.     setContactMediator(newMediator);
14.    createGui ();
15. }
16.  public void createGui(){
17.   setLayout(new BorderLayout () ) ;
18.    displayRegion = new JTextArea(10, 40);
19.     displayRegion.setEditable(false);
20.     add(new JScrollPane(displayRegion));
21. }
22.  public void contactChanged(Contact contact){
23.     diplayRegion.setText(
24.      "Contact\n\tName: " + contact.getFirstName() +
25.      " " + contact.getLastName() + "\n\tTitle;" +
26.      contact.getTitle() + "\n\tOrganization: " +
27.       contact.getOrganization());
28. }
29.  public void setContactMediator(ContactMediator newMediator){
30.     mediator = newMediator;
31. >
32.}
```

Класс ContactSelectorPanel (листинг 2.27) позволяет пользователю выбирать для отображения и редактирования объект класса Contact.

Листинг 2.27. ContactSelectorPanel.java

```
1. import java.awt.event.ActionEvent;
2. import java.awt.event.ActionListener;
3. import javax.swing.JComboBox;
4. import javax.swing.JPanel;
5.
6. public class ContactSelectorPanel extends JPanel implements ActionListener{
7.   private ContactMediator mediator;
8.   private JComboBox selector;
9.
10.  public ContactSelectorPanel(){
11.    createGui ();
12. }
13.  public ContactSelectorPanel(ContactMediator newMediator){
14.     setContactMediator(newMediator);
15.     createGui;
16. }
17.
18.  public void createGui(){
19.     selector = new JComboBox(mediator.getAllContacts());
20.     selector.addActionListener(this);
21.     add(selector);
22. }
23.
24.  public void actionPerformed(ActionEvent evt){
25.     mediator.selectContact((Contact)selector.getSelectedItem());
26. }
27.  public void addContact(Contact contact){
28.    selector.addItem(contact) ;
29.     selector.setSelectedItem(contact);
30. }
31.  public void setContactMediator(ContactMediator newMediator){
32.     mediator = newMediator;
33. }
34.}
```

Класс ContactEditorPanel (листинг 2.28) предоставляет интерфейс для редактирования полей выбранного контакта.

Листинг 2.28. ContactEditorPanel.java

```
1. import java.awt.BorderLayout;
2. import java.awt.GridLayout;
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5. import javax.swing.JButton;
6. import javax.swing.JLabel;
7. import javax.swing.JPanel;
8. import javax.swing.JTextField;
9. public class ContactEditorPanel extends JPanel implements ActionListener{
10.  private ContactMediator mediator;
11.  private JTextField firstName, lastName, title, organization;
12.  private JButton create, update;
13.
14.  public ContactEditorPanel(){
15.     createGui()
16.  }
17.  public ContactMediator(newMediator);
18.     setContactMediator(newMediator);
19.     createGui();
20.  }
21.  public void createGui(){
22.     setLayout(new BorderLayout());
23.
24.     JPanel editor = new JPanel ();
25.     editor.setLayout(new GridLayout(4, 2));
26.     editor.add(new JLabel.("First Name:"));
27.     firstName = new JTextField (20);
28.    editor.add(firstName);
29.     editor.add(new JLabel("Last Name:"));
30.     lastName = new JTextField(20);
31.     editor.add(lastName);
32.     editor.add(new JLabel("Title:"));
33.     title = new JTextField(20);
34.     editor.add(title);
35.     editor.add(new JLabel("Organization:"));
36.     organization = new JTextField(20);
37.     editor.add(organization);
38.     add(editor, BorderLayout.CENTER);
39.
40.     JPanel control = new JPanel();
41.     create = new JButton("Create Contact");
42.     update = new JButton("Update Contact");
43.     create.addActionListener(this);
44.     update.addActionListener(this);
45.     control.add(create);
46.     control.add(update);
47.     add(control, BorderLayout.SOUTH);
48.  }
49.  public void actionPerformed(ActionEvent evt){
50.     Object source = evt.getSource();
51.     if (source == create){
52.       createContact();
53.     }
54.     else if (source == update){
55.       updateContact();
56.     }
```

```
57. }
58.
59.  public void createContact (){
60.    mediator.createContact(firstName.getText(), lastName.getText(),
61.      title.getText(), organization.getText());
62. }
63.  public void updateContact(){
64.   mediator.updateContact(firstName.getText() , lastName.getText() ,
65.      title.getText(), organization.getText());
66. }
67.
68.  public void setContactFields(Contact contact){
69.     firstName.setText(contact.getFirstName());
70.    lastName.setText(contact.getLastName()) ;
71.   title.setText(contact.getTitle() ) ;
72.   organization.setText(contact.getOrganization() ) ;
73. }
74.  public void  setContactMediator(ContactMediator newMediator){
75.    mediator = newMediator;
76. }
77.}
```

Интерфейс ContactMediator (листинг 2.29) определяет набор методов для каждого компонента GUI и для методов, реализующих бизнес-правила (createContact, updateContact, selectContact и getAllContacts).

Листинг 2.29. ContactMediator.java

```
1. public interface ContactMediator{
2.   public void setContactDisplayPanel(ContactDisplayPanel displayPanel);
3.   public void setContactEditorPanel(ContactEditorPanel editorPanel);
4.   public void setContactSelectorPanel(ContactSelectorPanel selectorPanel);
5.   public void createContact(String firstName, String lastName, String title,
   String organization);
6.   public void updateContact(String firstName, String lastName, String title,
   String organization);
7.  public Contact [] getAllContacts () ;
8.   public void selectContact(Contact contact);
9. }
```

Реализация интерфейса ContactMediator представлена классом ContactMediatorImpl (листинг 2.30). Этот класс обеспечивает выбор контакта, а также содержит методы, посредством которых панели извещаются о внесении изменений пользовательского интерфейса.

Листинг 2.30. ContactMediatorImpl.java

```
1. import java.util.ArrayList;
2. public class ContactMediatorImpl implements ContactMediator{
3.   private ContactDisplayPanel display;
4.   private ContactEditorPanel editor:
5.   private ContactSelectorPanel selector;
6.   private ArrayList contacts = new ArrayList();
7.   private int contactIndex;
8.
9.   public void setContactDisplayPanel(ContactDisplayPanel displayPanel){
10.     display = displayPanel;
```

```
11. }
12.  public void setContactEditorPanel(ContactEditorPanel editorPanel){
13.     editor = editorPanel;
14.  }
15.  public void setContactSelectorPanel(ContactSelectorPanel){
16.     selector = selectorPanel;
17.  }
18.
19.  public void createContact(String firstName, String lastName, String title,
   String organization){
20.     Contact newContact = new ContactImpl(firstName, lastName, title,
   organization);
21.     addContact(newContact);
22.     selector.addContact(newContact);
23.     display.contactChanged(newContact);
24.  }
25.  public void updateContact(string firstName, String lastName, String title,
   String organization)(
26.     Contact updateContact = (Contact)contacts.get(contactIndex);
27.     if (updateContact != null)(
28.       updateContact.setFirstName(firstName);
29.       updateContact.setLastName(lastName);
30.       updateContact.setTitle(title);
31.       updateContact.setOrganization(organization);
32.       display.contactChanged(updateContact);
33.     }
34.  )
35.  public void selectContact(Contact contact){
36.     if (contacts.contains(contact)){
37.       contactIndex = contacts.indexOf(contact)
38.       display.contactChanged(contact)
39.       editor.setContactFields(contact);
40.     }
41.  )
42. public Contact [] getAllContacts () {
43.     return (Contact [])contacts.toArray(new Contact[1]);
44 . }
45.  public void addContact(Contact contact)(
46.    if (!contacts.contains(contact))(
47.      contacts.add(contact);
48.     )
49.  )
50.}
```

Класс `ContactMediatorImpl` взаимодействует с каждым из классов панелей по-разному. Так, обращаясь к классу `ContactDisplayPanel`, посредник для выполнения операций создания, обновления и выбора вызывает метод `contactChanged`. Обращаясь к классу `ContactSelectorPanel`, посредник предоставляет ему список контактов и метод `getAllContacts`, получает от него извещения о выбранном контакте и помещает новый объект класса `Contact` на панель при ее создании. Объект класса `ContactEditorPanel` вызывает при обращении к посреднику методы создания и обновления, а объект класса `ContactSelectorPanel` извещает посредника о работе панели выбора.

Memento

Также известен как Token, Snapshot

Свойства шаблона

Тип: поведенческий шаблон
Уровень: объект

Назначение

Сохраняет "моментальный список" состояния объекта, позволяющий такому объекту вернуться к исходному состоянию, не раскрывая своего содержимого внешнему миру.

Представление

В любом приложении имеются объекты, которым нужно на протяжении своего жизненного цикла время от времени где-то сохранять свою информацию. Часто такое сохранение обеспечивается путем совместного использования данных, однако в некоторых случаях объектам нужно сохранять внутреннюю информацию, не предназначенную для совместного доступа. Отправка такой информации другому объекту — не самое лучшее решение, так как оно противоречит правилам инкапсуляции. Действительно, если внутренние данные одного объекта становятся доступными для другого объекта, последний может не только прочитать их, но и, что еще хуже, модифицировать.

Это можно сравнить с посещением национального парка, занимающегося сохранением такого редкого животного, как американский лось. Роль объектов в данном случае играют особи лосей. Посетителям национального парка, несмотря на все их благие намерения, не разрешается забирать лосей домой. Все, что они могут, — это приобрести в киоске при входе в парк открытки и футболки с изображением американского лося.

Более удачный подход состоит в использовании специального объекта, содержащего все подлежащие сохранению данные. Вместо того чтобы отправлять для сохранения сами данные, отправляется такой объект-контейнер, с помощью которого можно восстановить оригинальный объект. Такой подход не позволяет другим объектам прочитать или модифицировать данные, так как они инкапсулированы в объекте-контейнере.

Данный подход реализуется шаблоном Memento, в котором определяется объект, используемый в качестве "шкатулки". Содержимое такое объекта доступно только его первоначальному владельцу, обеспечивая последнему возможность восстановления прежнего состояния.

Область применения

Шаблон Memento следует применять при *одновременном* наличии следующих обстоятельств.

- Необходимо создать "моментальный снимок" объекта.

- С помощью этого "моментального снимка" будет восстанавливаться исходное состояние объекта.
- Создание прямого интерфейса с объектом, обеспечивающего чтение его внутреннего состояния, нежелательно, так как нарушает принцип инкапсуляции и, кроме того, открывает внутреннюю логику объекта.

Описание

Если в приложении соблюдается принцип инкапсуляции, все объекты позволяют доступ к атрибутам, отражающим их внутреннее состояние, только через вызов соответствующих методов. Однако в некоторых случаях возникает необходимость передать текущее состояние другому объекту: например для того, чтобы впоследствии восстановить исходное состояние объекта (т.е. реализовать функцию отмены последней операции).

Лежащий на поверхности способ обеспечения такой возможности — это непосредственная передача состояния объекта. Но несмотря на кажущуюся простоту, этот способ имеет весьма серьезные недостатки.

- Раскрывается внутренняя структура объекта.
- Другой объект может по каким-то причинам изменить состояние сохраненного объекта.

Решение состоит в использовании объекта-контейнера, сохраняющего информацию о текущем состоянии объекта, основанного на шаблоне Memento. Таким образом, мы получаем объект Memento, который содержит информацию о текущем внутреннем состоянии объекта Originator. Только Originator может сохранять информацию в объекте Memento и извлекать ее оттуда. Для всех остальных объектов системы Memento — это рядовой объект.

Работу шаблона Memento можно сравнить с тем, как используется кредитная карта. Вам не нужно знать, как реализована логика, обеспечивающая обслуживание карты банкоматом, — достаточно знать, что она предназначена для снятия денег со счета. Так и шаблон Memento — достаточно знать, что он позволяет сохранять и восстанавливать состояние объекта.

Реализация

Диаграмма классов шаблона Memento представлена на рис. 2.9.

При реализации шаблона Memento обычно используются следующие классы.

- Originator. Создает объект класса Memento и использует его впоследствии для восстановления своего состояния.
- Memento. Статический класс, внутренний по отношению к классу Originator, в котором сохраняется информация о состоянии последнего. Объем сохраняемой информации определяется классом Originator, который, кроме того, обладает эксклюзивными правами чтения содержимого Memento. Сохраненная информация о состоянии недоступна никакому другому объекту, кроме Originator.

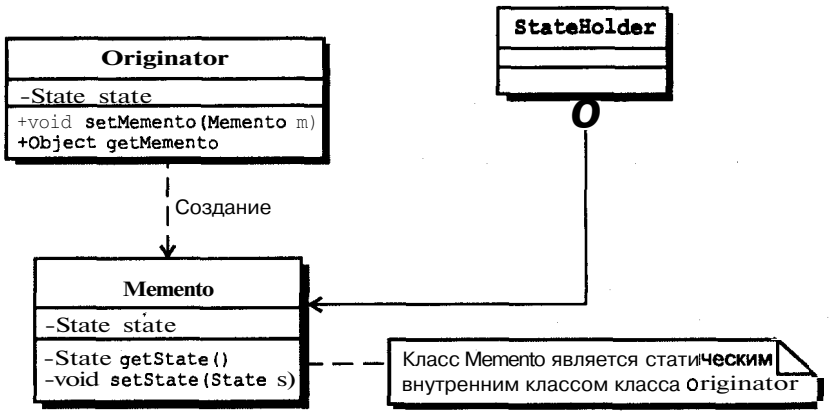


Рис. 2.9. Диаграмма классов шаблона Memento

- StateHolder. Объект, сохраняющий сведения о состоянии. Этому объекту ничего неизвестно об информации, сохраняемой внутри объекта Memento, — он лишь должен знать, что полученный им объект позволяет восстановить состояние объекта Originator.

Так как содержимое объекта Memento доступно лишь объекту Originator, лучше всего объявить класс Memento как общедоступный (public) внутренний класс класса Originator. При этом все методы класса Memento объявляются закрытыми (private), чтобы доступ к ним мог получить лишь сам класс Memento и класс, в котором он объявлен (т.е. Originator). Такое решение обеспечивает надежную инкапсуляцию данных.

Экземпляры внутреннего класса всегда связываются с экземпляром внешнего класса. Необходимость такой связи объясняется тем, что, внутренний класс постоянно имеет доступ ко внутренним переменным внешнего, класса. В данной ситуации это может привести к возникновению проблемы: объект класса Memento должен быть независимым от конкретного экземпляра класса Originator. Для обеспечения этой независимости класс Memento объявляется не просто внутренним классом, а статическим внутренним классом класса Originator.

Объекты класса Memento могут занимать значительный объем памяти, особенно если объекты класса Originator сохраняют в них всю информацию о своем состоянии и создают объекты класса Memento достаточно часто. В данной ситуации может помочь такая замена объектов Memento с тем, чтобы новый объект отражал лишь все изменения, произошедшие со времени создания предыдущего объекта. Порядок следования объектов Memento должен отслеживать объект StateHolder. В качестве примера использования такого подхода можно привести повышение по службе. Действительно, при каждом повышении размеры зарплаты и премий служащего изменяются в определенной пропорции к размеру предыдущих зарплат и премий. Если применяется такой подход, нет необходимости хранить сведения о выплатах служащему, начиная с первого дня его работы, — достаточно сохранять данные об изменениях в выплатах, произошедших со времени последнего повышения.

Достоинства и недостатки

Использование шаблона Memento влечет за собой следующие последствия.

- *Соблюдение принципа инкапсуляции.* Даже в тех случаях, когда состояние объекта Originator должно сохраняться клиентом за пределами класса Originator, клиент не может получить доступ к информации о хранимом им состоянии. Все, что у него есть, — это ссылка на объект класса Memento без какого-либо механизма доступа к находящейся внутри последнего информации. Кроме того, это позволяет упростить внутреннюю структуру клиента, так как ему не нужно знать внутреннее устройство класса Originator. Если клиент знает, как получить доступ к объекту Memento и как его использовать, он вполне может обойтись без сведений о работе класса Originator.
- *Упрощение класса Originator.* Предположим, что всю информацию о своих различных состояниях сохраняет сам класс Originator. Такой подход может очень скоро привести к "разбуханию" объекта и усложнению управления им. Гораздо проще возложить ответственность за сохранение состояний объекта на того, кому это нужно, т.е. на клиента. В таком случае класс Originator должен лишь знать, как создавать и использовать объекты класса Memento, полностью избавившись от сохранения своих состояний.
- *Накладные расходы на создание объектов-контейнеров.* Если в объектах-контейнерах хранится вся без исключения информация о состоянии объектов Originator, это может привести к резкому увеличению накладных расходов, связанных с выделением памяти для объектов Memento. Данный нежелательный эффект усиливается по мере увеличения размера данных объектов Originator. В таких случаях особенно важно применять сохранение не всей информации, а лишь той, которая изменилась после последнего сохранения. Но иногда, если класс Originator слишком велик, даже такая полумера не может компенсировать чрезвычайно высоких накладных расходов, что приводит к невозможности использования шаблона Memento.
- *Накладные расходы на хранение объектов-контейнеров.* Ответственность за управление объектами-контейнерами, полученными от объекта Originator, возложена на специальный объект `StateHolder`, представляющий собой хранилище информации о состоянии. Однако этот объект не имеет никакой информации о реальном размере данных, помещенных в объекты Memento. В тех случаях, когда разработчик не позаботился о сохранении минимального размера объектов Memento, ему придется столкнуться с проблемами, возникающими из-за накладных расходов на хранение объектов-контейнеров объектом StateHolder.

Варианты

Шаблон Memento может реализовываться в следующих вариантах.

- В тех случаях, когда по каким-то причинам класс Memento не может быть внутренним, нужно определить два интерфейса: `WideMemento` и NarrowMemento. Первый из них предназначен для того, чтобы класс Originator мог получить

доступ к объекту-контейнеру, в котором хранится информация о состоянии объекта Originator. Настройку состояния в объекте Memento лучше всего выполнять в конструкторе. Необходимо отметить, что использование интерфейса, а не абстрактного класса, влечет за собой добавление в класс Memento метода FactoryMethod.

Второй интерфейс необходим классу StateHolder, а также другим клиентам. Если в интерфейсе не объявлен ни один метод, он становится ненужным, а вызывающие объекты ссылаются на объекты класса Memento, как на объекты класса Object.

- Если нужно расширить класс Originator, не меняя класса Memento, методы можно не объявлять закрытыми, а обеспечить к ним доступ через пакет. Это позволит создать подклассы класса Originator, сохранив возможность использования того же класса Memento.

Родственныешаблоны

К родственным относятся следующие шаблоны.

- Command (стр. 71). Шаблон Command использует объекты-контейнеры для отслеживания состояния выполняемых операций с целью обеспечения возможности их отмены.
- State (стр. 120). В большинстве случаев объекты, созданные при использовании шаблона State, задействуют также и шаблон Memento.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом RunPattern, приведен в разделе "Memento" на стр. 403 Приложения А.

Практически все подсистемы РШ-приложения сохраняют в том или ином объеме информацию о своем состоянии. Эту информацию можно сохранять с помощью шаблона Memento, как показано в данном разделе на примере адресной книги. Класс AddressBook (листинг 2.31) представляет собой коллекцию адресов, что делает его одним из кандидатов на сохранение информации о состоянии.

Листинг 2.31. AddressBook.java

```
1. import java.util.ArrayList;
2. public class AddressBook{
3.   private ArrayList contacts = new ArrayList();
4.
5.   public Object getMemento(){
6.     return new AddressBookMemento(contacts);
7.   }
8.   public void setMemento(Object object){
9.     if (object instanceof AddressBookMemento){
```

```
10.      AddressBookMemento memento = (AddressBookMemento)object;
11.      contacts = memento.state;
12.    }
13. >
14.
15. private class AddressBookMemento{
16.   private ArrayList state;
17.
18.   private AddressBookMemento(ArrayList contacts){
19.     this.state = contacts;
20.    }
21. }
22.
23. public AddressBook(){ }
24. public AddressBook(ArrayList newContacts){
25.   contacts = newContacts;
26. }
27.
28. public void addContact(Contact contact){
29.   if (!contacts.contains (contact)){
30.    contacts.add(contact) ;
31.    }
32. }
33. public void removeContact(Contact contact){
34.   contacts.remove(contact);
35. }
36. public void removeAllContacts (){
37.   contacts = new ArrayList();
38. }
39. public ArrayList getContacts(){
40.    return contacts;
41. }
42. public String toString(){
43.  return contacts. toString ();
44. }
45.}
```

Для сохранения информации о состоянии класса AddressBook, представленной в виде внутреннего объекта ArrayList, который предназначен для хранения объектов Address, используется внутренний по отношению к нему класс AddressBookMemento. Доступ к объекту-контейнеру можно получить, используя методы getMemento и setMemento класса AddressBook. Необходимо отметить, что класс AddressBookMemento объявлен как закрытый внутренний класс, состоящий лишь из одного закрытого конструктора. Это делает невозможным использование другими объектами информации о состоянии, хранящейся в контейнере, или ее изменение даже в тех случаях, когда объект AddressBookMemento сохраняется за пределами класса AddressBook. Данное свойство класса AddressBookMemento полностью соответствует цели, стоящей перед шаблоном Memento: генерация объекта, содержащего "моментальный снимок" состояния, который не может быть модифицирован ни одним посторонним объектом системы.

Observer

Также известен как Publisher-Subcriber

Свойствашаблона

Тип: поведенческий шаблон
Уровень: компонент

Назначение

Предоставляет компоненту возможность гибкой рассылки сообщений интересующим его получателям.

Представление

Как быть в тех случаях, когда один из множества объектов изменился, а эти изменения остались без внимания со стороны других объектов?

Предположим, что в PIM-приложении нужно организовать совместный доступ пользователей к информации. Эта возможность будет полезной, например, для координации регулярного собрания членов клуба "ОРЗ" ("Объединенные расшифровщики зебр"). Для того чтобы сообщить членам клуба о месте, дате и времени проведения собрания, можно использовать объект Appointment. Однако как гарантировать, что при изменении, к примеру, времени проведения собрания измененная информация о запланированном событии будет доведена до всех заинтересованных лиц? (Ведь на собрание никто не придет!)

Конечно, можно вести список всех членов клуба и разослать каждому из них обновленную информацию. Но такой метод больше подходит для тех собраний, явка на которые строго обязательна. Если же речь идет о рядовом собрании, которое наверняка многие члены клуба решат не посещать, организация рассылки — пустая трата времени и сил. Кроме того, такое решение неэффективно и с технической точки зрения, так как в тех случаях, когда членов клуба очень много, это влечет за собой перегрузку средств коммуникации.

Лучше всего оставить выбор того, нужна ли им информация о планируемом собрании, за самими пользователями. В этом случае объект Appointment сохраняется на сервере. Если тому или иному члену клуба нужно получить обновленную информацию о планируемом собрании, они подключаются к серверу. При каждом обновлении объекта Appointment сервер рассылает новую информацию подключенным к нему пользователям.

Данный подход, обеспечивающий высокий уровень гибкости с точки зрения организации рассылки обновленной информации, закреплен в виде шаблона под названием Observer. Такое название объясняется тем, что шаблон предполагает существование некоего центрального объекта, за которым наблюдают, как в обсерватории, другие объекты, заинтересованные в получении от первого какой-то информации. Потребовав от наблюдающих объектов, чтобы они устанавливали сеансы связи с центральным объектом, можно значительно снизить накладные расходы на коммуника-

цию. Последнее обстоятельство объясняется тем, что сеансы связи устанавливаются лишь теми объектами, которые действительно заинтересованы в получении обновленной информации.

Область применения

Шаблон Observer лучше всего применять в тех случаях, когда система обладает следующими свойствами.

- Существует, как минимум, один объект, рассылающий сообщения.
- Имеется не менее одного получателя сообщений, причем их количество и состав может изменяться во время работы приложения, а также различаться у разных экземпляров приложения.

Данный шаблон часто применяют в ситуациях, в которых отправитель сообщений не должен или не хочет знать, что делают получатели с предоставленной им информацией. Его задача — лишь разослать информацию всем, кому она нужна.

Описание

Некоторые отправители сообщений, называемые также *генераторами сообщений* (message producers), работают в соответствии с простой моделью одноранговых коммуникаций, создавая сообщения, которые предназначаются строго определенным *получателям сообщений* (message receiver). В таких случаях организовать обработку событий очень просто. Однако бывают генераторы сообщений, чье поведение далеко не так просто. Отправка сообщения может повлечь за собой переменное количество последовательных ответных реакций, в которых задействуются как генератор, так и один или более получателей.

Рассмотрим, например, обработку адреса клиента. Можно легко представить бизнес-модель, в которой изменение адреса влечет за собой множество ответных реакций системы. Нужно обновить информацию о клиенте, сведения о его подписке, и возможно, текущие заказы. Теоретически может измениться даже такая информация, как стоимость доставки и ставка налога с продаж. Именно в таких ситуациях и применяется шаблон Observer.

В соответствии с этим шаблоном, генераторы сообщений (наблюдаемые компоненты) рассылают сообщения, которые генерируют события в системе. Эти события обрабатываются одним или несколькими получателями сообщений (компоненты-наблюдатели). Наблюдаемые компоненты отвечают за доставку событий всем заинтересованным наблюдателям, т.е. всем тем наблюдателям, которые установили сеансы связи с наблюдаемым компонентом. Интерфейс наблюдения позволяет наблюдаемым компонентам указать все возможные события и даже сообщить какие-то детали наблюдающим компонентам.

Шаблон Observer можно представить как решение, основанное на доставке информации сервером (server-push). Сервер (в данном случае — наблюдаемый объект, или генератор событий) контактирует с заинтересованным наблюдателем в тех случаях, когда возникает то или иное событие.

Шаблон Observer может оказаться полезным в самых разных приложениях. Так как извещения рассылаются только тем элементам, которые идентифицировали себя в качестве заинтересованных получателей, последние могут реагировать на полученную информацию так, как они считают нужным. Данный подход хорошо применим к любой модели, в которой, во-первых, изменения одного компонента могут приводить к изменению других компонентов, а во-вторых, поведение объектов может настраиваться во время работы приложения.

С точки зрения бизнес-модели шаблон Observer полезен в тех случаях, когда в модели реализуются сложные операции изменения, удаления или обновления информации. Гибкая природа шаблона позволяет применять его для рассылки информации как некоторым, так и всем элементам модели.

Работу шаблона Observer можно сравнить с тем, что происходит в жизни молодого человека и девушки во время ухаживания. Каждый из них (наблюдаемый) имеет одного или нескольких друзей (наблюдатели), которые проявляют интерес к тому, как развиваются отношения между влюбленными. По мере того, как в жизни молодого человека и его возлюбленной происходят новые события (т.е. свидания, заканчивающиеся определенными успехами или неудачами), они делятся своими впечатлениями с друзьями, сообщая определенные детали о своих взаимоотношениях. Друзья реагируют на полученные сообщения по-разному: советами, поздравлениями, сочувствием, а то и завистью.

Реализация

Диаграмма классов шаблона Observer представлена на рис. 2.10.

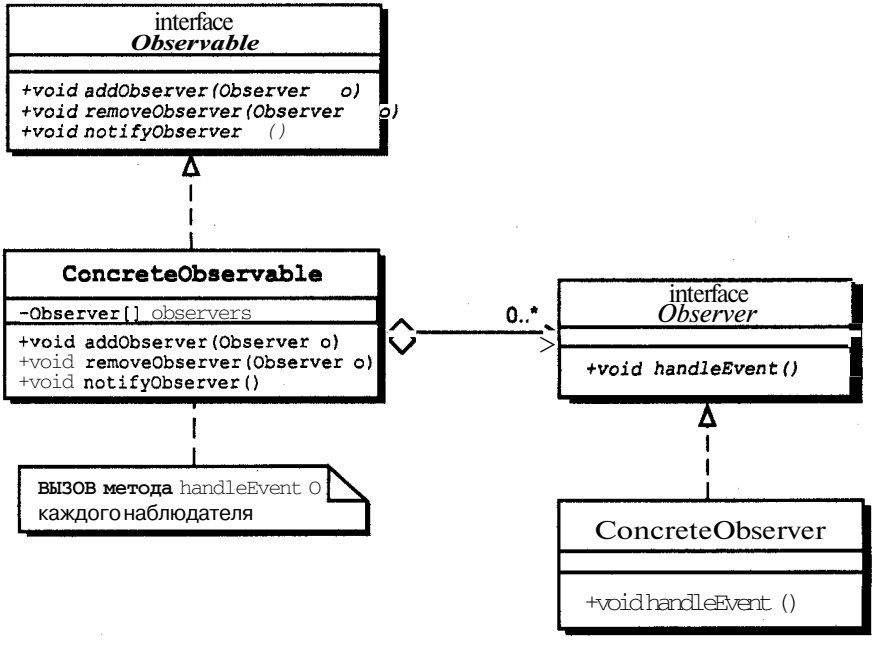


Рис. 2.10. Диаграмма классов шаблона Observer

При реализации шаблона Observer обычно используются следующие классы.

- Observable. Интерфейс, который определяет, как наблюдатели-клиенты могут взаимодействовать с наблюдаемым объектом. Среди методов интерфейса должны быть методы добавления и удаления наблюдателей, а также несколько методов (или хотя бы один метод) извещения, предназначенные для рассылки информации от наблюдаемого объекта клиентам.
- ConcreteObservable. Класс, который содержит реализацию всех методов, определенных в интерфейсе Observable. Этот класс должен обеспечить работу с коллекцией объектов класса Observer.

  Методы извещения копируют или клонируют список таких объектов и, последовательно перебирая их, вызывают определенные методы наблюдателей каждого объекта Observer.
- Observer. Интерфейс, с помощью которого наблюдаемый объект взаимодействует с клиентами.
- ConcreteObserver. Реализует интерфейс Observer и определяет в каждом реализуемом методе, как реагировать на сообщения, полученные от объекта класса Observable.

Обычно задача подключения конкретных наблюдателей к наблюдаемому объекту возлагается на специальные подсистемы приложения.

Достоинства и недостатки

Помимо гибкости, шаблон Observer примечателен еще и тем, что не требует, чтобы наблюдаемый объект имел сложную структуру. Это означает, что для его применения разработчику не придется тратить много времени на написание соответствующего программного кода. Кроме того, этот шаблон обладает неоспоримыми достоинствами в следующих аспектах.

- *Тестирование.* Можно написать дополнительный объект-наблюдатель, предназначенный для отображения на экране всего, что делает наблюдаемый объект.
- *Поэтапная разработка.* Шаблон позволяет добавлять в систему дополнительные объекты-наблюдатели по мере их разработки.

  Самые большие затруднения при использовании этого шаблона вытекают из реализуемой в нем модели, построенной на обмене сообщениями. Можно использовать как специальную, так и универсальную стратегию рассылки сообщений, каждая из которых имеет свои недостатки.
- *Универсальная стратегия.* Поскольку сообщения носят универсальный характер, сложнее определить, что конкретно происходит с наблюдаемым объектом. Кроме того, универсальная стратегия характеризуется избыточным потоком сообщений — некоторые события, пересылаемые наблюдателям, последними никак не обрабатываются, а лишь влекут за собой увеличение накладных расходов. Наконец универсальная стратегия требует дополнительных затрат на разработку классов наблюдателей, так как им нужно анализировать сообщения и вычленять из них необходимую информацию.

- *Специальная стратегия.* Сообщения, направленные на решение узкой задачи обмена информацией наблюдаемого объекта с наблюдателями, выдвигают повышенные требования относительно программирования первого, поскольку он должен генерировать последовательности извещений при возникновении определенных условий. Это может повлечь за собой и увеличение сложности объектов-наблюдателей, так как им придется обрабатывать сообщения разных типов.

Варианты

Шаблон Observer может реализовываться в нескольких вариантах, основанных на характере взаимоотношений наблюдаемого объекта и наблюдателей.

- *Один или несколько наблюдателей.* В зависимости от роли, которую играют наблюдаемые компоненты, они могут реализовываться таким образом, чтобы поддерживать только одного наблюдателя.
- *Многопоточные наблюдаемые компоненты.* Если наблюдаемый объект многопоточный, он может поддерживать очередь сообщений, а также предоставить дополнительные возможности, такие как приоритеты сообщений и перекрытие сообщений.
- *Самостоятельное получение сообщений клиентами (client pull).* Хотя шаблон ориентирован на доставку информации сервером, его можно модифицировать таким образом, чтобы в какой-то степени обеспечить самостоятельное получение сообщений клиентами. В данном варианте наблюдаемый объект обычно извещает наблюдателей о том, что имело место какое-то событие. Если наблюдателям нужны более детальные сведения, они вступают в контакт с наблюдаемым объектом, вызывая метод, который запрашивает дополнительную информацию о событии.

Родственные шаблоны

К родственным можно отнести шаблон Proxy (стр. 209). Если нужно обеспечить распределенные коммуникации, этот шаблон часто применяется для взаимодействия объектов Observer и Observable.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, *приведен в разделе "Observer" на стр.* **407** *Приложения* **А.**

В данном примере показано, как наблюдаемый объект рассылает всем наблюдателям информацию об обновленном состоянии объекта `Task`.

Необходимо отметить, что в любом программном коде GUI Java для обработки сообщений обычно используется именно шаблон Observer. Разрабатывая класс, реализующий интерфейс вида `ActionListener`, вы тем самым создаете класс наблюдателя.

Регистрация этого класса в компоненте с помощью метода addActionListener связывает наблюдателя с наблюдаемым элементом, роль которого выполняет компонент GUIJava.

В данном примере наблюдаемый элемент представлен классом Task, который модифицируется GUI. Класс TaskChangeObservable (листинг 2.32) отслеживает изменения в объекте класса Task с помощью методов addTaskChangeObserver и removeTaskChangeObserver.

Листинг 2.32. TaskChangeObservable.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class TaskChangeObservable{
A.  private ArrayList observers = new ArrayList ();
5.
6.   public void addTaskChangeObserver(TaskChangeObserver observer){
7.      if  (!observers.contains(observer)){
8.        observers.add(observer);
9.      }
10. }
11.  public void removeTaskChangeObserver(TaskChangeObserver observer){
12.     observers.remove(observer);
13. }
14.
15.  public void selectTask(Task task){
16.     Iterator elements = observers.iterator();
17.     while (elements.hasNext()){
18.       ((TaskChangeObserver)elements.next()).taskSelected(task);
19.     }
20. }
21.  public void addTask(Task task){
22.     Iterator elements = observers.iterator();
23.     while (elements.hasNext()){
24.       ((TaskChangeObserver)elements.next()).taskAdded(task);
25.     )
26. }
27.  public void updateTask(Task task){
28.     Iterator elements = observers.iterator();
29.     while (elements.hasNext()){
30.       ((TaskChangeObserver) elements.next()).taskChanged(task);
31.     }
32. )
33.}
```

Класс TaskChangeObservable содержит методы selectTask, updateTask и addTask. Эти методы отвечают за отправку извещения о любых изменениях классу Task.

Каждый наблюдатель должен реализовать интерфейс TaskChangeObserver (листинг 2.33), чтобы класс TaskChangeObservable мог вызвать соответствующий метод наблюдателя. Если клиенту понадобилось вызвать метод addTask класса TaskChangeObservable, например, наблюдаемый объект будет последовательно вызывать методы taskAdded всех своих наблюдателей.

Листинг 2.33. TaskChangeObservable.java

```
1.  public interface TaskChangeObserver{
2.  public void taskAdded(Task task) ;
3.  public void taskChanged(Task task);
4.  public void taskSelected(Task task);
5.  }
```

Класс ObserverGui (листинг 2.34) предназначен в рассматриваемом примере для организации графического пользовательского интерфейса. Именно он создает объект класса TaskChangeObservable. Кроме того, он создает три панели TaskEditorPanel (листинг 2.35), TaskHistoryPanel (листинг 2.36) и TaskSelectorPanel (листинг 2.37), которые реализуют интерфейс TaskChangeObserver, а потом связывает с ними объект класса TaskChangeObservable. После такого связывания последний получает возможность эффективно отправлять сообщения об обновлении всем трем панелям GUI.

Листинг 2.34. ObserverGui.java

```
1. import java.awt.Container;
2. import java.awt.event.WindowAdapter;
3. import java.awt.event.WindowEvent;
4. import javax.swing.BoxLayout;
5. import javax.swing.JFrame;
6. public class ObserverGui{
7.   public void createGui(){
8.     JFrame mainFrame = new JFrame("Observer Pattern Example");
9.     Container content = mainFrame.getContentPane();
10.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
11.    TaskChangeObservable observable = new TaskChangeObservable();
12.    TaskSelectorPanel select = new TaskSelectorPanel(observable);
13.    TaskHistoryPanel history = new TaskHistoryPanel();
14.    TaskEditorPanel edit = new TaskEditorPanel(observable);
15.    observable.addTaskChangeObserver(select);
16.    observable.addTaskChangeObserver(history);
17.    observable.addTaskChangeObserver(edit);
18.    observable.addTask(new Task()));
19.    content.add(select);
20.    content.add(history) ;
21.    content.add(edit) ;
22.    mainFrame.addWindowListener(new WindowCloseManager());
23.    mainFrame.pack():
24.    mainFrame.setVisible(true) ;
25. }
26.
27.  private class WindowCloseManager extends WindowAdapter{
28.    public void windowClosing(WindowEvent evt)(
29.      System.exit(0);
30.    }
31. }
32.}
```

Листинг 2.35. TaskEditorPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JLabel;
4. import javax.swing.JTextField;
5. import javax.swing.JButton;
6. import java.awt.event.ActionEvent;
7. import java.awt.event.ActionListener;
8. import java.awt.GridLayout;
9. public class TaskEditorPanel extends JPanel implements ActionListener,
   TaskChangeObserver{
10.  private JPanel controlPanel, editPanel;
11.  private JButton add, update, exit;
12.  private JTextField taskName, taskNotes, taskTime;
13.  private TaskChangeObservable notifier;
14.  private Task editTask;
15.
16.  public TaskEditorPanel(TaskChangeObservable newNotifier){
17.     notifier = newNotifier;
18.    createGui () ;
19. }
20.  public void createGui (){
21.     setLayout(new BorderLayout());
22.     editPanel = new JPanel();
23.     editPanel.setLayout(new GridLayout(3, 2)) ;
24.     taskName = new JTextField(20);
25.     taskNotes = new JTextField(20);
26.     taskTime = new JTextField(20);
27.     editPanel.add(new JLabel("Task Name"));
28.     editPanel.add(taskName);
29.     editPanel.add(new JLabel("Task Notes"));
30.     editPanel.add(taskNotes);
31.     editPanel.add(new JLabel("Time Required"));
32.     editPanel.add(taskTime);
33.
34.     controlPanel = new JPanel();
35.      add = new JButton("Add Task");
36.     update = new JButton("Update Task");
37.     exit = new JButton("Exit") ;
38.     controlPanel.add(add) ;
39.     controlPanel.add(update) ;
40.     controlPanel.add(exit) ;
41.     add.addActionListener(this) ;
42.     update.addActionListener(this) ;
43.      exit.addActionListener(this);
44.     add(controlPanel, BorderLayout.SOUTH) ;
45.     add(editPanel, BorderLayout.CENTER);
46. }
47.  public void setTaskChangeObservable(TaskChangeObservable newNotifier){
48.     notifier = newNotifier;
49. }
50.  public void actionPerformed(ActionEvent event){
51.     Object source = event.getSource();
52.     if (source == add) {
53.       double timeRequired = 0.0;
54.      try{
55.           timeRequired = Double.parseDouble(taskTime.getText());
56.       }
57.        catch (NumberFormatException exc){}
58.     notifier.addTask(new Task(taskName.getText(), taskNotes.getText(),
   timeRequired));
```

```
59.     }
60.     else if (source == update){
61.      editTask.setName(taskName.getText() ) ;
62.       editTask.setNotes(taskNotes.getText());
63.       try{
64.          editTask.setTimeRequired(Double.parseDouble(taskTime.getText() ) ) ;
65.       }
66.       catch (NumberFormatException exc){}
67.      notifier.updateTask(editTask) ;
68.     }
69.     else if (source == exit)(
70.      System.exit(0) ;
71.     }
72.
73. }
74. public void task taskAdded(Task task){ }
75. public void taskChanged(Task task){ }
76. public void taskSelected(Task task){
77.    editTask = task;
78.    taskName.setText(task.getName() ) ;
79.    taskNotes.setText(task.getNotes() ) ;
80.    taskTime.setText("" + task.getTimeRequired()) ;
81. }
82.}
```

Листинг 2.36. TaskHistoryPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JScrollPane;
4. import javax.swing.JTextArea;
5. public class TaskHistoryPanel extends JPanel implements TaskChangeObserver{
6.  private JTextArea displayRegion;
7.
8.  public TaskHistoryPanel(){
9.     createGui();
10. }
11. public void createGui(){
12.    setLayout(new BorderLayout()) ;
13.    displayRegion = new JTextArea(10, 40);
14.    displayRegion.setEditable(false);
15.    add(new JScrollPane(displayRegion));
16. }
17. public void taskAdded(Task task)(
18.   displayRegion.append("Created task" + task + "\n");
19. }
20. public void taskChanged(Task task){
21.   displayRegion.append("Updated task " + task + "\n");
22. )
23. public void taskSelected(Task task){
24.   displayRegion.append("Selected task " + task + "\n");
25. }
26.}
```

Листинг 2.37. TaskSelectorPanel.java

```
1. import java.awt.event.ActionEvent;
2. import java.awt.event.ActionListener;
3. import javax.swing.JPanel;
4. import javax.swing.JComboBox;
5. public class TaskSelectorPanel extends JPanel implements ActionListener,
   TaskChangeObserver{
6.   private JComboBox selector = new JComboBox();
7.   private TaskChangeObservable notifier;
8.   public TaskSelectorPanel(TaskChangeObservable newNotifier){
9.     notifier = newNotifier;
10.    createGui();
11.  }
12.  public void createGui(){
13.    selector = new JComboBox();
14.    selector.addActionListener(this);
15.    add(selector);
16.  }
17.  public void actionPerformed(ActionEvent evt){
18.    notifier.selectTask((Task)selector.getSelectedItem());
19.  }
20.  public void setTaskChangeObservable(TaskChangeObservable newNotifier){
21.    notifier = newNotifier;
22.  }
23.
24.  public void taskAdded(Task task){
25.    selector.addItem(task) ;
26.  }
27.  public void taskChanged(Task task){ }
28.  public void taskSelected(Task task){ }
29.}
```

Одно из свойств, присущих шаблону Observer, состоит в том, что класс наблюдаемого объекта Observable использует стандартный интерфейс для своих наблюдателей (в данном случае он называется TaskChangeObserver). Это означает не только то, что шаблон Observer является более универсальным, чем, скажем, шаблон Mediator, но и то, что объекты-наблюдатели могут получать определенное количество ненужных им сообщений. Например, класс TaskEditorPanel, как это видно из листинга 2.35, не выполняет никаких действий при вызове его методов `taskAdded` и `taskChanged`.

State

Также известен как Objects for States

Свойства шаблона

Тип: поведенческий шаблон
Уровень: объект

Назначение

Обеспечивает изменение поведения объекта во время выполнения программы.

Представление

Часто требуется полностью изменить поведение приложения при достижении определенными переменными заданного значения. Например, всем известно, что при работе с текстовыми редакторами нужно время от времени сохранять текст в файле. Современные текстовые редакторы позволяют сохранить документ только в том случае, если в нем были сделаны какие-то изменения. После того, как документ был сохранен, считается, что все, что отображается на дисплее, точно соответствует содержимому файла. С этого момента функция сохранения становится недоступной, так как в ней до внесения пользователем новых изменений нет никакой необходимости.

Реализация подобных механизмов принятия решения в виде отдельных методов усложняет как разработку текста программы, так и его понимание. Как правило, усложнение выражается в использовании множества сложных и объемных структур вида `if-else`. Возможно также представление состояния объекта в виде константы с определенным значением, что требует использования во всех методах объемных операторов вида `switch-case`, которые часто дублируют друг друга.

Любой объект можно представить в виде комбинации определенного состояния и поведения. Состояние объекта сохраняется в его атрибутах, а поведение определяется его методами. Шаблон State позволяет динамически менять поведение объекта. Это достигается путем делегирования вызовов всех методов, зависящих от значений определенных переменных, объекту `State`. Такой объект также представляет собой комбинацию состояния и поведения, поэтому, меняя объекты `State`, мы получаем разные виды поведения. Теперь в методах конкретного класса `State` можно не использовать операторы `if-else` и `switch-case`, так как объект `State` определяет поведение только для одного состояния.

Областьприменения

Шаблон State рекомендуется использовать в следующих случаях.

- Поведение объекта зависит от его состояния, которое часто меняется.
- В методах используются объемные условные операторы, организующие изменение поведения в зависимости от состояния.

Описание

Без шаблона State трудно реализовать объекты, которые должны менять свое поведение в зависимости от текущего состояния. Как уже говорилось, попытка реализации таких объектов без шаблона State зачастую сводится к использованию констант, представляющих различные состояния, и весьма объемных операторов передачи управления, помещаемых внутри нескольких методов объекта. При этом большинство таких методов очень похожи друг на друга, так как все они реализуют одну и ту же логику — определение текущего состояния.

Рассмотрим такой простой объект реального мира, как дверь. Какие операции мы обычно выполняем с дверью? Дверь можно открыть или закрыть, или, иными словами, перевести в одно из двух состояний: "открыто" или "закрыто". Вызов метода за-

крытия для уже закрытой двери не даст никаких результатов, но вызов этого же метода для открытой двери приведет к изменению ее состояния на "закрыто".

Диаграмма состояний объекта "дверь" приведена на рис. 2.11.

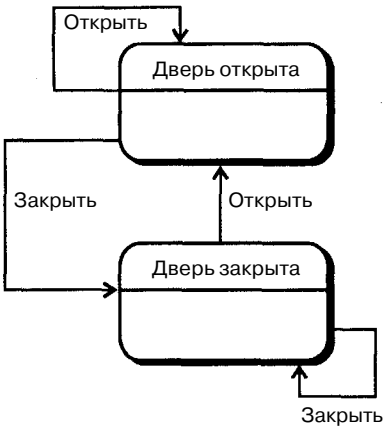


Рис. 2.11. Диаграмма состояний двери

Таким образом, текущее состояние двери оказывает влияние на ее поведение, заставляя ее реагировать по-разному на одну и ту же команду.

Реализация

Диаграмма классов шаблона State представлена на рис. 2.12.

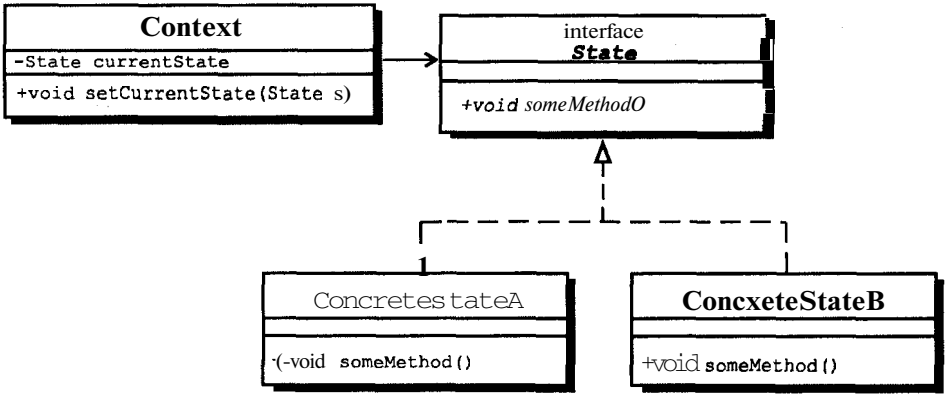


Рис. 2.12. Диаграмма классов шаблона State

При реализации шаблона State обычно используются следующие классы.

- Context. Содержит ссылку на текущее состояние, представляя собой интерфейс для других клиентов. Делегирует вызовы всех связанных с тем или иным состоянием методов текущему объекту класса `State`.

- State. Определяет все методы, зависящие от состояния объекта.
- ConcreteState. Реализует интерфейс State, воплощая конкретную модель поведения для определенного состояния.

Определять переход между состояниями может как Context, так и ConcreteState — это шаблоном State не оговаривается. Когда количество состояний заранее известно и ограничено определенным значением, лучше всего поместить логику переключения состояний в интерфейс Context.

С другой стороны, размещение этой логики в подклассе класса State обеспечивает большую гибкость. При данном подходе переходы состояний выполняются каждым объектом класса `State`. При этом объект определяет, в какое состояние нужно перейти (т.е. устанавливает ссылку на следующий объект класса State), а также в каких случаях и когда выполняется переход. Данный подход значительно упрощает задачи изменения части схемы переходов состояний и добавления новых состояний в эту схему. Недостатком этого подхода является то, что каждый класс, реализующий интерфейс State, зависит от других классов, поскольку любая реализация класса State должна знать о существовании, как минимум, еще одного экземпляра этого класса. Когда реализация класса State определяет необходимость выполнения перехода, экземпляр класса Context должен обеспечить установку нового состояния в соответствующем контексте.

Объекты состояния можно создавать по мере необходимости и заранее.

- Создание объектов `State` по мере необходимости чаще всего применяют в тех случаях, когда состояние меняется редко. Кроме того, этот способ является единственно возможным в ситуациях, когда в начале работы приложения количество различных состояний неизвестно. Данный подход позволяет избежать создания больших объектов состояний, когда нет твердой уверенности в том, что в этом возникнет необходимость.
- В большинстве случаев объекты состояния создаются заранее, при запуске приложения. При этом создаются все необходимые экземпляры класса `State`, а ресурсы, требуемые для создания множества объектов, расходуются лишь единожды. Данный подход имеет смысл применять при частом изменении состояний, когда велика вероятность того, что новый объект состояния понадобится очень скоро.

Достоинства и недостатки

Шаблон имеет State как достоинства, так и некоторые недостатки.

- *Разделение поведения на основе состояния.* Этим обеспечивается более четкое представление о разных моделях поведения объекта. Когда объект находится в определенном состоянии, достаточно обратить внимание на соответствующий подкласс State, чтобы понять все возможные варианты поведения в данном состоянии.
- *Структурирование системы и упрощение ее понимания.* Широкораспространенным альтернативным по отношению к шаблону State решением является использование констант и оператора условного перехода, с помощью которого определяются различные варианты реакций на возникающие состояния. Это не самое

лучшее решение, так как оно приводит к значительному дублированию кода, поскольку многие методы содержат практически совпадающие операторы. Если в такую систему нужно добавить новое состояние, приходится вносить изменения во все методы класса Context, добавляя новый элемент в каждый оператор switch-case. Это не только требует значительных усилий, но и повышает вероятность появления в системе ошибок. Если же система использует шаблон State, в ней достаточно создать новую реализацию класса State.

- *Явное представление перехода состояния.* Когда для представления состояния используются константы, изменение состояния довольно легко перепутать с присвоением значения переменной, так как с точки зрения синтаксиса обе операции выглядят идентично. Если же используются объекты класса State, переход состояния очень легко идентифицировать и отличить от других операций.
- *Совместный доступ к состояниям.* Если подклассы класса State содержат лишь определения поведения без каких-либо переменных, они могут выполнятся в соответствии с шаблоном Flyweight (стр. 196). Передачу состояния объектам, выполненным в соответствии с шаблоном Flyweight, обеспечивает класс Context. Это позволяет уменьшить количество объектов в системе.
- *Использование большого количества классов.* Увеличение количества классов, как правило, считается недостатком. В соответствии с шаблоном State для каждого возможного состояния создается не менее одного класса. Но если рассмотреть недостатки альтернативного способа (пространные условные операторы в методах), становится ясно, что большое количество классов — это скорее достоинство шаблона State, чем его недостаток, поскольку распределение функциональности между классами позволяет значительно улучшить понимание логики программы.

Варианты

Одна из главных сложностей при реализации шаблона State — это выбор объекта, который должен управлять изменением состояния. Возможные варианты, заключающиеся в выборе класса Context или подклассов State, уже обсуждались выше. Третий вариант состоит в создании таблицы, в которой определены все возможные варианты состояний. В данном варианте разработка кода, обеспечивающего переход состояний, сводится к написанию операции выборки из таблицы по заданным критериям.

Достоинством же данного варианта является определенная упорядоченность. Для того чтобы изменить критерии перехода, достаточно внести коррективы в табличные данные, не изменяя кода приложения. Однако здесь имеются и свои недостатки.

- Поиск в таблице чаще всего оказывается менее эффективной операцией, чем вызов метода.
- Если логика переходов представлена в виде таблицы, ее сложнее понять.

Главное отличие данного подхода состоит в том, что шаблон State сосредоточен на моделировании поведения в зависимости от состояния, тогда как табличный метод — на переходах между различными состояниями.

Но если объединить эти два подхода, можно получить табличную модель с поддержкой шаблона State. В этом случае переходы хранятся схемой хеширования

(HashMap), но вместо того, чтобы создавать таблицу для каждого состояния, создается объект HashMap для каждого метода интерфейса State. Это обстоятельство объясняется тем, что объект, представляющий следующее состояние, скорее всего будет иметь методы, практически не совпадающие с методами текущего объекта состояния.

В схеме хеширования, представленной объектом HashMap, прежнее состояние используется как ключ, а новое — как значение. Теперь значительно проще добавить новое состояние — достаточно лишь определить новый класс и изменить с его помощью схему хеширования. Данный вариант подробнее представлен в подразделе "Пример" данного раздела.

Родственныешаблоны

К родственным можно отнести следующие шаблоны.

- Flyweight (стр. 196). С помощью шаблона Flyweight можно организовать совместное использование объектов State.
- Singleton (стр. 54). Большинство объектов State выполняется в соответствии с шаблоном Singleton, особенно в тех случаях, когда используется шаблон Flyweight.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`приведен в разделе "State" на стр. 413Приложения А.

Объекты States лучше всего определять в виде внутренних классов, так как это обеспечивает им тесную связь с основным классом, а также прямой доступ к атрибутам последнего. В приведенном ниже примере показано, как это работает на практике.

Стандартный подход к сохранению информации в приложениях состоит в том, чтобы выполнять такие операции только по мере необходимости, т.е. после внесения изменений. Когда изменения внесены, но файл не сохранен, его состояние называют "плохим" (dirty). Это означает, что содержимое открытого файла может отличаться от содержимого ранее сохраненной версии. Файл, который был сохранен и в который после сохранения не вносились никакие изменения, называют "хорошим" (clean). В "хорошем" состоянии содержимое открытого и сохраненного файлов идентично.

В данном примере показано, как использовать шаблон State для обновления объектов Appointment PIM-приложения с сохранением их по мере необходимости в файле. Диаграмма перехода состояний для файла представлена на рис. 2.13.

Два класса состояний (CleanState и DirtyState) реализуют интерфейс State (листинг 2.38). Эти классы отвечают за определение следующего состояния (в данном случае это несложно, так как имеется всего лишь два состояния).

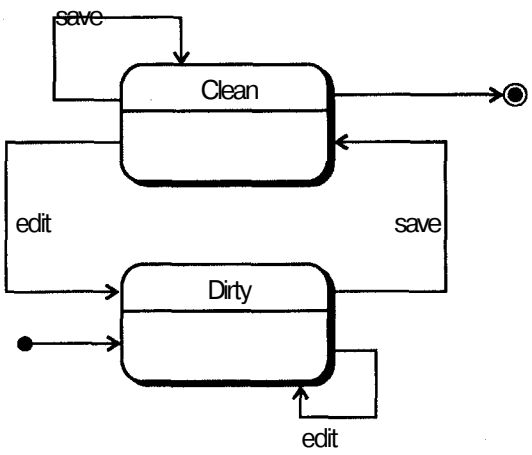


Рис.2.13. Диаграмма перехода состояний для файла

В интерфейсе State определены два метода — save и edit, которые в случае необходимости вызываются классом CalendarEditor (листинг 2.39).

Листинг 2.38. State.java

```
1. public interface State{
2.   public void saved;
3.  public void edit () ;
4. }
```

Класс CalendarEditor управляет коллекцией объектов Appointment.

Листинг 2.39. CalendarEditor.java

```
1. import java.io.File;
2. import java.util.ArrayList;
3. public class CalendarEditor)
4.    private State currentstate;
5.    private File appointmentFile;
6.    private ArrayList appointments = new ArrayList();
7.    private static final String DEFAULT_APPOINTMENT_FILE = "appointments.ser";
8.
9.    public CalendarEditor(){
10.      this(DEFAULT_APPOINTMENT_FILE);
11.   }
12.   public CalendarEditor(String appointmentFileName))
13.      appointmentFile = new File(appointmentFileName);
14.      try)
15.        appointments = (ArrayList)FileLoader.loadData(appointmentFile);
16.      }
17.      catch (ClassCastException exc){
18.        System.err.println("Unable to load information. The file does not
   contain a list of appointments.");
19.      }
20.      currentstate = new CleanState ();
21.}
```

```
22.
23.  public void saved (
24.    currentState.save() ;
25.  }
26.
27.  public void edit(){
28.    currentState.edit() ;
29.  }
30.
31.  private class DirtyState implements State{
32.     private State nextstate;
33.
34.     public DirtyState(State nextstate){
35.       this.nextState = nextstate
36.     }
37.
38.     public void saved)
39.       FileLoader.storeData(appointmentFile, appointments);
40.       currentState = nextstate;
41.     }
42.     public void edit(){ }
43.  }
44.
45.  private class CleanState implements State{
46.     private State nextstate = new DirtyState(this);
47.
48.     public void save(){ }
49.     public void edit(){ currentState = nextstate; }
50.  )
51.
52.  public ArrayList getAppointments(){
53.     return appointments;
54.  }
55.
56.  public void addAppointment(Appointment appointment);
57.     if (!appointments.contains(appointment)){
58.       appointments.add(appointment);
59.     }
60.  }
61.  public void removeAppointment(Appointment appointment){
62.     appointments.remove(appointment);
63.  }
64.}
```

Класс StateGui (листинг 2.40) обеспечивает интерфейс редактирования для событий, запланированных с помощью редактора CalendarEditor. Нужно заметить, что объект GUI хранит ссылку на CalendarEditor, с помощью которой делегирует редактору права на внесение изменений или сохранение. Это позволяет редактору выполнять требуемые операции и обновлять в случае необходимости свое состояние.

Листинг 2.40. StateGui.java

```
1. import java.awt.Container;
2. import java.awt.BorderLayout;
3. import java.awt.event.ActionListener;
4. import java.awt.event.WindowAdapter;
5. import java.awt.event.ActionEvent;
6. import java.awt.event.WindowEvent;
7. import javax.swing.BoxLayout;
8. import javax.swing.JButton;
```

```
9. import javax.swing.JComponent;
10. import javax.swing.JFrame;
11. import javax.swing.JPanel;
12, import javax.swing.JScrollPane;
13. import javax.swing.JTable;
14. import javax.swing.table.AbstractTableModel;
15. import java.util.Date;
16,public class StateGui implements ActionListener{
17,  private JFrame mainFrame;
18.  private JPanel controlPanel, editPanel;
19,  private CalendarEditor editor;
20,  private JButton save, exit;
21,
22.  public StateGui(CalendarEditor edit){
23,     editor = edit;
24,  }
25,
26,  public void createGui(){
27,     mainFrame = new JFrame("State Pattern Example");
28,     Container content = mainFrame.getContentPane();
29,     content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
30,
31.     editPanel = new JPanel();
32,     editPanel.setLayout(new BorderLayout());
33.     JTable appointmentTable = new JTable(new StateTableModel((Appointment
   []editor.getAppointments().toArray(newAppointment[1])));
34.     editPanel.add(new JScrollPane(appointmentTable));
35.     content.add(editPanel);
36.
37.     controlPanel = new JPanel();
38.     save = new JButton("Save Appointments");
39.     exit = new JButton("Exit");
40,     controlPanel.add(save);
41,     controlPanel.add(exit);
42,     content.add(controlPanel);
43,
44,     save.addActionListener(this);
45,     exit.addActionListener(this);
46,
47,     mainFrame.addWindowListener(new WindowCloseManager());
48,     mainFrame.pack();
49,     mainFrame.setVisible(true);
50,  }
51.
52.
53.  public void actionPerformed(ActionEvent evt){
54.     Object originator = evt.getSource();
55.     if (originator == save){
56.       saveAppointments();
57.     }
58.     else if (originator == exit){
59.       exitApplication();
60.     }
61.  }
62.
63.  private class WindowCloseManager extends WindowAdapter{
64.     public void windowClosing(WindowEvent evt){
65.       exitApplication();
66.     }
67.  }
68.
69.  private void saveAppointments(){
70.     editor.save();
```

```
71.  }
72.
73.  private void exitApplication(){
74.     System.exit(0);
75.  }
76.
77.  private class StateTableModel extends AbstractTableModel{
78.     private final String [] columnNames = {
79.       "Appointment", "Contacts", "Location", "Start Date", "End Date" };
80.     private Appointment [] data;
81.
82.     public StateTableModel(Appointment [] appointments);
83.       data = appointments;
84.     }
85.
86.     public String getColumnName(int column){
87.       return columnNames[column] ;
88.     }
89.     public int getRowCount(){ return data.length; }
90.     public int getColumnCount(){ return columnNames.length; }
91.     public Object getValueAT(int row, int column){
92.       Object value = null;
93.       switch(column){
94.            case 0: value = data[row].getReason();
95.                 break;
96.            case 1: value = data[row].getContacts();
97.                 break;
98.            case 2: value = data[row].getLocation();
99.                 break;
100.           case 2: value = data[row].getStartDate();
101.                break;
102.           case 4: value - data[row].getEndDate();
103.                break;
104.      }
105.      return value;
106.    }
107.    public boolean isCellEditable(int row, int column){
108.      return ((column ==0) | | (column ==2)) ? true : false;
109.    }
110.    public void setValueAt(Object value, int row, int column){
111.      switch(column){
112.           case 0: data[row].setReason((String)value);
113.                editor.edit();
114.                break;
115.           case 1:
116.                break;
117.           case 2: data[row].setLocation(new LocationImpl((String)value));
118.                editor.edit();
119.                break;
120.           case 3:
121.                break;
122.           case 4:
123.                break;
124.      }
125.   }
126. }
127.}
```

Strategy

Также известен как Policy

Свойства шаблона

Тип: поведенческий шаблон
Уровень: компонент

Назначение

Предназначен для определения группы классов, которые представляют собой набор возможных вариантов поведения. Это дает возможность гибко подключать те или иные наборы вариантов поведения во время работы приложения, меняя его функциональность "на ходу".

Представление

Предположим, в PIM-приложении имеется список контактов. По мере увеличения этого списка у пользователя неизбежно возникает потребность в средствах сортировки списка и получения сводной информации о контактах.

Для обеспечения такой возможности необходимо создать класс коллекции, который бы выполнял хранение контактной информации в памяти, сортировку объектов и получение сводной информации о контактах. Такое решение на какой-то период может оказаться достаточным, но со временем "всплывают" и другие проблемы. Самая серьезная из них состоит в том, что такое решение нельзя модифицировать или расширить, не прилагая значительных усилий. Всякий раз, когда нужно внести новую функциональность в методы сортировки и получения сводной информации, необходимо переписывать класс коллекции. Более того, по мере увеличения количества различных поддерживаемых методов сортировки или получения сводной информации, увеличивается размер и сложность программного кода коллекции, что усложняет отладку и сопровождение.

А что если разработать *серию* классов, в которой каждый класс обеспечивает выполнение определенного способа сортировки или подведения итогов? Класс коллекции делегирует соответствующие задачи одному из этих классов и таким образом получает возможность использовать различные подходы, или стратегии, для выполнения нужных ему операций без усложнения собственного кода и других присущих традиционному подходу недостатков.

Идеи, положенные в основу шаблона Strategy, основываются на таких характеристиках объектов, как состояние и поведение. Заменив один объект другим, можно изменить поведение. И хотя при этом увеличивается количество классов, каждый из них остается простым в сопровождении, а полученная в результате система — очень гибкой.

Областьприменения

Шаблон Strategy рекомендуется использовать в следующих случаях.

- Имеется несколько возможных способов выполнения операций.
- При разработке программы не всегда известно, какой способ будет наилучшим во время работы приложения.
- Необходимо обеспечить простоту добавления новых способов выполнения операций.
- Необходимо сохранить простоту исходного кода независимо от объема добавляемой функциональности.

Описание

Нередки ситуации, в которых одну и ту же задачу можно решить несколькими способами. К одной из таких задач относится, например, сортировка — для ее выполнения можно применить множество различных хорошо документированных алгоритмов, таких как "быстрая сортировка" или "метод пузырька". Можно также выполнять сортировку по нескольким полям или по нескольким критериям. Когда в распоряжении объекта имеется несколько способов решения стоящей перед ним задачи, он становится слишком сложным и трудно управляемым. Достаточно только представить объем работы, которую необходимо проделать, чтобы написать исходный текст класса, предназначенного для представления и сохранения документа в совершенно разных форматах: здесь и обычный текст, и документ StarOffice, и файл Postscript и т.д. Чем больше возрастают количество и сложность поддерживаемых форматов, тем больше требуется прилагать усилий для сопровождения исходного текста класса.

Чтобы достигнуть баланса между гибкостью и сложностью, в таких случаях можно использовать шаблон Strategy. Этот шаблон отделяет различные типы поведения от объекта, представляя их в виде отдельной иерархии классов. Объект может использовать тот или иной тип поведения в зависимости от конкретных обстоятельств. Возвращаясь к примеру с документом, можно разработать один класс для сохранения документа в каждом формате, а поведение класса при сохранении будет определяться суперклассом или интерфейсом.

Шаблон Strategy очень полезен в тех случаях, когда нужно обеспечить управление набором алгоритмов, таких, как алгоритмы сортировки или поиска. Кроме того, его можно очень эффективно применять для управления запросами к базам данных, используя различные подходы к выполнению запросов, представлению результатов или выбирая различные стратегии кэширования данных. В реальном мире шаблон Strategy также иногда используется в тех случаях, когда нужно представить различные способы выполнения бизнес-функций. Размещения заказа на изготовление рабочей станции, например, может происходить в соответствии с шаблоном Strategy, если состав и комплектация блоков и узлов заказанной станции существенно отличаются от стандартных.

Как и шаблон State (см. стр. 120), шаблон Strategy разделяет компонент на отдельные группы классов, делегируя функции, которые представляют поведение компонента, отдельному набору обработчиков.

Достоинства и недостатки

В связи с тем, что каждый тип поведения представляется в виде отдельного класса, шаблон Strategy позволяет получить более удобную в сопровождении систему. При добавлении новых типов поведения становится гораздо проще расширить модель, так как нет необходимости в переписывании всего приложения.

Основная проблема при использовании шаблона Strategy состоит в том, чтобы правильно выбрать способ представления всех имеющихся типов поведения. Это не такая простая задача, как может показаться на первый взгляд, поскольку способ вызова всех стратегий объектом должен быть унифицированным. Поэтому разработчику сначала нужно выбрать такой способ, который, с одной стороны, подошел бы всем реализациям класса, а с другой — соответствовал бы каждой конкретной стратегии.

Реализация

Диаграмма классов шаблона Strategy представлена на рис. 2.14.

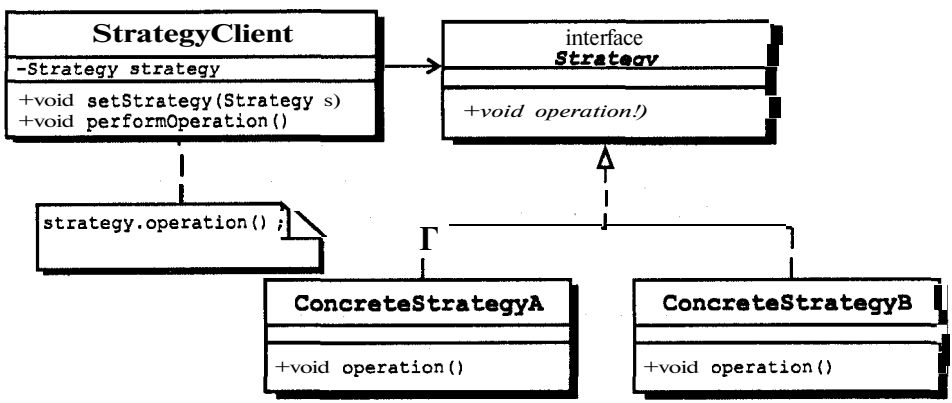


Рис. 2.14. Диаграмма классов шаблона Strategy

При реализации шаблона Strategy обычно используются следующие классы.

- StrategyClient. Класс, который использует различные стратегии для выполнения стоящих перед ним задач. Он содержит ссылку на текущий экземпляр класса Strategy, а также имеет метод, обеспечивающий замену текущего экземпляра другим.
- Strategy. Интерфейс, определяющий все методы, которые доступны для использования классом StrategyClient.
- ConcreteStrategy. Класс, реализующий интерфейс Strategy и использующий конкретные наборы правил внутри каждого определенного интерфейсом метода.

Варианты

Отсутствуют.

Родственные шаблоны

К родственным относятся следующие шаблоны.

- Singleton (стр. 54). Реализации шаблона Strategy иногда представляются в виде одиночных объектов или статических ресурсов.
- Flyweight (стр. 196). Иногда объекты стратегий выполняются в соответствии с шаблоном Flyweight, чтобы уменьшить накладные расходы на их создание.
- Factory Method (стр. 42). Шаблон Strategy в некоторых случаях определяется в виде некоторого механизма с тем, чтобы класс, работающий со стратегиями, мог применять новые реализации шаблона Strategy без внесения модификаций в остальные части приложения.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Strategy" на стр. 422 Приложения А.

Многие коллекции PIM-приложения могут использовать определенные способы организации хранящихся в них элементов и получения сводной информации о них. В данном примере показано, как с помощью шаблона Strategy можно обеспечить получение сводной информации об элементах коллекции ContactList (листинг 2.41), которые представляют собой объекты класса `Contact`.

Листинг 2.41. ContactList.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public class ContactList implements Serializable{
4.   private ArrayList contacts = new ArrayList ();
5.   private SummarizingStrategy summarizer;
6.
7.   public ArrayList getContacts (){ return contacts; }
8.   public Contact [] getContactsAsArray(){ return (Contact []
    (contacts.toArray(newContact [1])); }
9.
10.  public void setSummarizer(SummarizingStrategy newSummarizer){ summarizer =
    newSummarizer; }
11.  public void setContacts(ArrayList newContacts){ contacts = newContacts; }
12.
13.  public void addContact(Contact element){
14.     if (!contacts.contains(element)){
15.       contacts.add(element) ;
16.     }
11.  }
18.  public void removeContact(Contact element){
19.     contacts.remove(element);
20.  }
21.
22.  public String summarize(){
```

```
23.     return summarizer.summarize(getContactsAsArray());
24. }
25.
26. public String [] makeSummarizedList(){
27.     return summarizer.makeSummarizedList(getContactsAsArray());
28. }
29.}
```

Класс ContactList имеет два метода, с помощью которых коллекция может получать сводную информацию о хранящихся в ней объектах Contact — summarize и makeSummarizedList. Оба метода делегированы интерфейсу SummarizingStrategy (листинг 2.42),ссылкана экземпляр которого устанавливается методом setSumma rizer класса коллекции.

Листинг 2.42. SummarizingStrategy.java

```
1. public interface SummarizingStrategy{
2.   public static final String EOL_STRING =
   System.get Property ("line.separator");
3.   public static final String DELIMITER = ":";
4.   public static final String COMMA = ",";
5.   public static final String SPACE = " ";
6.
7.   public String summarize(Contact [] contactList);
8.   public String [] makeSummarizedList(Contact [] contactList);
9. }
```

SummarizingStrategy — это интерфейс, который определяет два делегированных ему метода summarize и makeSummarizedList. Данный интерфейс соответствует интерфейсу Strategy в иерархии классов шаблона Strategy. В рассматриваемом примере класс ConcreteStrategy представлен двумя классами: NameSummarizer (листинг 2.43) и OrganizationSummarizer (листинг 2.44). Оба класса обеспечивают получение сводной информации об элементах списка контактов, однако каждый из них представляет собственный набор информации и группирует данные по-своему.

Класс NameSummarizer возвращает лишь имена и фамилии контактных лиц, указывая первой фамилию. Сравнение выполняется внутренним классом NameComparator, который выполняет сортировку всех элементов списка в алфавитном порядке сначала по фамилиям, а затем по именам.

Листинг 2.43. NameSummarizer.java

```
1. import java.text.Collator;
2. import java.util.Arrays;
3. import java.util.Comparator;
4. public class NameSummarizer implements SummarizingStrategy{
5.   private Comparator comparator = new NameComparator();
6.
7.   public String summarize (Contact [] contactList) {
8.     StringBuffer product = new StringBuf fer () ;
9.     Arrays.sort(contactList, comparator);
10.    for (int i = 0; i< contactList.length; i++){
11.      product.append(contactList[i] .getLastName());
12.      product.append(COMMA);
```

```
13.       product.append(SPACE);
14.     product.append(contactList[i].getFirstName() ) ;
15.       product.append(EOL_STRING);
16.     >
17.     return product.toString();
18.  }
19.
20.  public String!] makeSummarizedList(Contact [] contactList){
21.    Arrays.sort(contactList, comparator);
22.     String [] product = new String[contactList.length]
23.     for (int i = 0; i < contactList.length; i++){
24.      product[i] = contactList[i].getLastName() + COMMA + SPACE +
25.           contactList[i].getFirstName()  + EOL_STRING;
26.     }
27.     return product;
28.  }
29.
30.  private class NameComparator implements Comparator{
31.     private Collator textComparator = Collator.getInstance();
32.
33.     public int compare(Object o1, Object o2){
34.       Contact c1, c2;
35.       if ((o1 instanceof Contact) && (o2 instanceof Contact)){
36.            c1 = (Contact)o1;
37.            c2 = (Contact)o2;
38.            int compareResult = textComparator.compare(c1.getLastName(),
   c2.getLastName());
39.            if (compareResult == 0){
40.                 compareResult = textComparator.compare(c1.getFirstName(),
   c2.getFirstName());
41.            }
42.            return compareResult;
43.       }
44.       else return textComparator.compare(o1,o2);
45.     }
46.
47.     public boolean equals(Object o){
48.       return textComparator.equals(o);
49.     }
50.  }
51.}
```

Класс OrganizationSummarizer представляет контактную информацию, отсортированную по организациям с указанием имени и фамилии контактного лица. Объект класса Comparator обеспечивает сортировку информации в алфавитном порядке сначала по организациям, а затем по фамилиям.

Листинг 2.44. OrganizationSummarizer.java

```
1. import java.text.Collator;
2. import java.util.Arrays;
3. import java.util.Comparator;
4. public class OrganizationSummarizer implements SummarizingStrategy{
5.   private Comparator comparator = new OrganizationComparator()
6.
7.   public String summarize(Contact [] contactList))
8.      StringBuffer product = new StringBuffer()
9.     Arrays.sort(contactList, comparator);
10.     for (int i = o; i < contactList.length; i++){
11.      product.append(contactList[i].getOrganization() ) ;
```

```
12.       product.append(DELIMITER);
13.       product.append(SPACE);
14  .  product.append(contactList[i].getFirstName()  )  ;
15.       product.append(SPACE);
16.       product.append(contactList[i].getLastName());
17.       product.append(EOL_STRING);
18.     }
19.     return product.toString();
20.  }
21.
22.  public String [] makeSummarizedList(Contact  [] contactList){
23.    Arrays.sort(ContactList, comparator);
24.    String  [] product = new String[contactList.length];
25.    for  (int i = o; i < contactList.length; i++){
26.      product[i]  = contactList[i].getOrganization()  + DELIMITER + SPACE +
27.           contactList[i].getFirstName()  + SPACE +
28.           contactList[i].getLastName()  + EOL_STRING;
29.     }
30.     return product;
31.  }
32.
33.  private class OrganizationComparator implements Comparator{
34.    private Collator textComparator = Collator.getInstance ();
35.
36.    public int compare(Object o1, Object o2){
37.      Contact c1,c2;
38.      if  ((o1 instanceof Contact)  &&  (o2 instanceof Contact))}
39.           c1 =  (Contact)o1;
40.           c2 =  (Contact)o2;
41.           int compareResult = textComparator.compare(c1.getOrganization(),
                                                          c2.getOrganization());
42.           if  (compareResult == 0){
43.           compareResult = textComparator.compare(c1.getLastName(),
   c2.getLastName());
44.            }
45.           return compareResult;
46.      }
47.      else return textComparator.compare(o1, o2);
48.     }
49.
50.     public boolean equals(Object o){
51.      return textComparator.equals(o) ;
52.     }
53.  }
54.}
```

Visitor

Свойства шаблона

Тип: поведенческий шаблон
Уровень: от компонента до системы

Назначение

Обеспечивает простой и удобный в эксплуатации способ выполнения тех или иных операций для определенного семейства классов. Это достигается путем централизации с помощью данного шаблона возможных вариантов поведения, что позволяет модифицировать или расширять их, не затрагивая классы, на которые распространяются эти варианты поведения.

Представление

Предположим, что нужно обеспечить поддержку в PIM-приложении средств управления проектами, которые бы позволяли оформлять наряды, выполнять анализ рисков и производить оценку временных затрат. Любой проект средней сложности можно представить с помощью следующих классов.

- `Project`. Корень иерархической системы классов проекта, представляющий весь проект в целом.
- `Task`. Отдельный этап проекта.
- `DependentTask`. Этап проекта, для завершения которого необходимо завершение других этапов, от которых он зависит.
- `Deliverable`. Элемент или документ, получаемый в результате проекта.

Для того чтобы явно идентифицировать эти классы, как составные части одной модели, они организовываются на основе общего интерфейса `ProjectItem`.

Пока что все понятно. Но как, к примеру, запрограммировать возможность оценки общей стоимости проекта? Вычисления, по-видимому, зависят от конкретного типа элемента `ProjectItem`. В интерфейсе определяется метод `getCost`, который должен выполнять вычисления стоимости соответствующего этапа проекта. Таким образом, можно вычислить стоимость любого элемента, входящего в структуру проекта.

Можно, например, применить следующий подход к распределению функциональности.

- *Проект*. Нет смысла выполнять какие-либо специальные операции, поскольку стоимость всего проекта складывается из стоимости всех его этапов.
- *Обычный этап*. Стоимость определяется объемом человеко-часов, необходимых для его выполнения.
- *Зависимый этап*. Стоимость определяется как для обычного этапа, но с учетом дополнительных расходов, связанных с необходимостью координации этапов.
- *Результат*. Стоимость определяется стоимостью материалов и издержками на производство.

Вроде бы все просто и понятно. Однако как быть в тех случаях, когда нужно определить время, необходимое для завершения проекта, или оценить степень его рискованности? Если использовать такой же подход, как и для определения стоимости, усложнится задача сопровождения исходного кода. А еще хуже то, что при последующем добавлении средств поддержки подобной функциональности снова придется

дописывать дополнительные методы для многих классов приложения. Таким образом, по мере роста возможностей приложения классы становятся все объемнее и сложнее, что, помимо прочего, значительно затрудняет понимание их назначения и принципов работы.

Кроме того, при таком подходе затрудняется и собственно реализация функциональности. Действительно, при использовании локальных методов для подсчета стоимости, определения временных затрат или оценки степени риска необходимо каким-то образом обеспечить накопление и передачу промежуточных результатов, поскольку каждый метод принадлежит определенному объекту проекта. Большинство программистов в такой ситуации разработают какой-либо механизм передачи подобной информации по дереву проекта, надеясь, что им не придется столкнутся с отладкой этого механизма.

Альтернативой здесь может стать шаблон Visitor. Он позволяет, определив лишь один класс, например, CostProjectVisitor, выполнить с его помощью все вычисления, необходимые для определения стоимости всего проекта или какой-то его части. Вместо того чтобы подсчитывать стоимость внутри объектов ProjectItem, достаточно лишь передать их классу Visitor, который подсчитает все издержки для заданного этапа.

При таком подходе нет необходимости в определении в классе ProjectItem метода getCost. Вместо него определяется универсальный метод acceptVisitor, который использует класс вида ProjectVisitorдля вызова нужного метода последнего. Например, метод acceptVisitor объекта Task вызывает метод visitTask класса Visitor. Если экземпляр класса Visitor представлен, например, объектом класса CostProjectVisitor, метод visitTask будет вычислять стоимость этапа проекта, представленного вызывающим объектом класса Task.

Описанное архитектурное решение предоставляет множество преимуществ. Самое важное из них состоит в том, что это решение значительно облегчает задачу добавления новых операций. Для того чтобы добавить функциональность, определяющую время выполнения проекта, достаточно лишь разработать новый класс TimeProjectVisitor, обладающий всеми методами, необходимыми для определения времени на каждом этапе проекта. При этом исходный код объектов проекта остается неизменным, так как в нем уже реализована поддержка вызова универсальных методов, определенных на уровне класса ProjectVisitor.

Помимо этого, шаблон Visitor позволяет централизованно хранить результаты вычислений. Так, возвращаясь к примеру с классом CostProjectVisitor, можно хранить промежуточные результаты, получаемые в ходе подсчета стоимости, в самом экземпляре класса Visitor. Централизованный оценочный код также упрощает настройку базовых вычислений. Использование шаблона Visitor позволяет легко добавлять дополнительную функциональность, такую, например, как определение весовых характеристик для каждого этапа.

Область применения

Шаблон Visitor рекомендуется использовать в следующих случаях.

- Система содержит группу взаимосвязанных классов.

- Необходимо выполнять некоторые нетривиальные операции над определенной частью этих классов или всеми классами.
- Операции для разных классов должны выполняться по-разному.

Описание

Шаблон Visitor позволяет вычленить подобные операции из группы классов и собрать эти операции в одном классе. Главная причина выбора этого шаблона — обеспечение удобства сопровождения исходного кода, так как в некоторых ситуациях выполнение операций в разных классах, а не в одном делает задачу сопровождения слишком сложной. Тогда весьма полезно использовать шаблон Visitor, так как он обеспечивает универсальный механизм для выполнения операций над группами классов.

Для реализации данного шаблона требуется, чтобы все классы вида Element, над которыми будут выполняться операции, имели в своем составе метод вида accept, который должен вызываться в тех случаях, когда класс вида Visitor должен выполнить операцию над классом Element. В качестве параметра такому методу передается ссылка на экземпляр класса Visitor. В реализации класса Element в методе accept реализуется вызов метода вида visit класса Visitor для собственного типа класса. Каждая реализация класса Visitor должна содержать реализацию метода visit для подтипа класса Element.

В перечень систем, которые могут успешно использовать шаблон Visitor, входят системы, имеющие сложные правила конфигурации. В качестве практического примера можно привести транспортное средство, выступающее в роли товара. При покупке автомобиля могут иметь значение десятки различных факторов, причем многие из них могут влиять на такие показатели, как цена, ставка кредита или размер страхового взноса.

Реализация

Диаграмма классов шаблона Visitor представлена на рис. 2.15.
При реализации шаблона Visitor обычно используются следующие классы.

- Visitor. Абстрактный класс или интерфейс, который определяет метод visitor для каждого класса ConcreteElement.
- ConcreteVisitor. Класс, представляющий определенную операцию, выполняемую в системе. Реализует все методы, определенные классом или интерфейсом Visitor для реализации конкретной операции или конкретного алгоритма.
- Element. Абстрактный класс или интерфейс, представляющий некий объект, над которым выполняет операции класс Visitor. Он должен содержать определение метода accept, которому в качестве параметра передается ссылка на экземпляр класса Visitor.
- ConcreteElement. Конкретная реализация класса или интерфейса Element, представляющая собой некоторую сущность системы. В этом классе содержится реализация метода accept, определенного на уровне Element. Этот метод вызывает соответствующий метод visit, определенный на уровне Visitor.

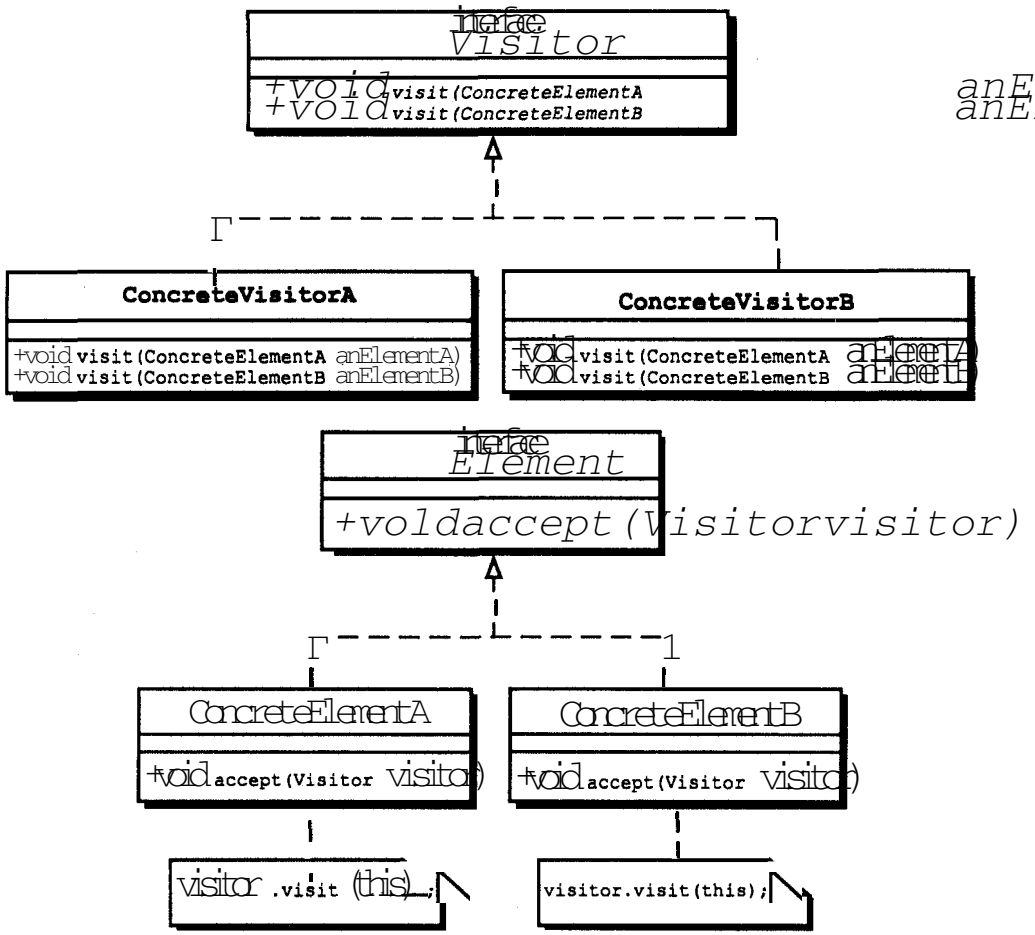


Рис. 2.15. Диаграмма классов шаблона Visitor

При реализации шаблона Visitor необходимо учитывать такой фактор, как наличие перекрытых методов. Шаблон использует перекрытие методов при работе с методами `visit`. На рис. 2.15 показано, что в интерфейсе `Visitor` определены два метода `visit`, каждый со своим параметром. В аспекте языка программирования эти методы не имеют между собой ничего общего.

Хотя реализации методов `accept` в каждом классе `ConcreteElement` близки по форме одна другой (а в некоторых случаях просто совпадают), их нельзя вынести в суперкласс. Это объясняется тем, что при вызове из суперкласса методу `visit` в качестве параметра будет передаваться тип суперкласса, даже если действительный тип экземпляра будет соответствовать типу класса `ConcreteElement`.

Достоинства и недостатки

Благодаря своей структуре шаблон Visitor значительно облегчает задачу добавления к системе новой функциональности. При реализации шаблона создаются все механизмы, необходимые для поддержки других подобных операций, в реализации которых впоследствии может возникнуть необходимость. Затем для того чтобы добавить новую функциональность, достаточно лишь создать один класс, реализующий интерфейс Visitor, и воплотить в нем нужную функциональность.

Достоинства шаблона Visitor проявляются в том, что он позволяет централизовать функциональный код, выполняющий определенную операцию. Помимо того, что данный подход упрощает задачу расширения или модификации кода по прошествию длительного времени, он также обеспечивает поддержание системы в контролируемом состоянии. Так как обычно для работы с каждым элементом структуры используется один и тот же объект класса Visitor, он представляет собой очень удобное централизованное хранилище для накопления собранных данных или для хранения промежуточных результатов.

Недостатком шаблона является то, что цепочка классов Element практически лишена такого свойства, как гибкость. При любом добавлении или модификации иерархии классов Element степень вероятности того, что это повлечет за собой необходимость модификации структуры кода классов Visitor, всегда остается высокой. Добавление любого дополнительного класса требует добавления нового метода в интерфейс Visitor и реализации этого метода в каждом классе ConcreteVisitor.

Кроме того, данный шаблон нарушает или, во всяком случае, серьезно искажает объектно-ориентированный принцип инкапсуляции кода. Действительно, шаблон Visitor получает ссылку на программный код, который применяется к объектам, внешним по отношению к классу объекта `Visitor`, и переносит ее в другое место.

Классам `Element` не нужно знать детали устройства конкретной реализации класса Visitor, но классам ConcreteVisitor *нужно* знать детали устройства классов Element. Это объясняется тем, что для выполнения заданной функции или производства вычислений объекту Visitor часто приходится использовать методы классов Element, необходимые ему для решения своей задачи.

Варианты

Как и в случае со многими другими шаблонами, `Element` и Visitor могут выполняться как в виде абстрактных классов, так и в виде интерфейсов языка Java.

Кроме того, разработчик должен выбрать способ применения шаблона Visitor к коллекции, содержащей объекты Element. Необходимо подчеркнуть, что механизм, построенный на основе шаблона Visitor, ничего не знает о структуре классов Element, над которыми он выполняет заданные операции. Поэтому класс ConcreteVisitor можно использовать одинаково эффективно при работе с простой коллекцией, связным списком, деревом или иерархией.

Хотя в некоторых реализациях шаблона код перемещения по объектам помещается внутри класса ConcreteVisitor, шаблон этого не требует. Для этих целей вполне можно использовать какой-нибудь внешний класс, например Iterator, который и будет выполнять перемещение по коллекции. Поэтому в тех случаях, когда это требуется, можно использовать шаблон Visitor вместе с шаблоном Iterator.

Родственные шаблоны

Помимо перечисленных ниже шаблонов, которые можно отнести к числу родственных для шаблона Visitor, разработчик может использовать и любые другие шаблоны, обеспечивающие перебор элементов коллекций.

- Interpreter (стр. 79). С помощью шаблона Visitor можно централизовать операции, выполняемые при интерпретации.
- Iterator (стр. 87). Шаблон Iterator применяется для перемещения по коллекциям.
- Composite (стр. 171). Шаблон Composite предоставляет шаблону Visitor механизм перемещения по древовидным структурам.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом *RunPattern, приведен в разделе "Visitor" на стр.* ***429*** *Приложения* ***А.***

Шаблон Visitor часто используется в тех случаях, когда по ходу операций, выполняемых над структурами большого объема, необходимо вычислять промежуточные и итоговые результаты. В данном примере показано, как с помощью шаблона Visitor обеспечивается выполнение расчета общей стоимости проекта.

Для представления элементов проекта используют четыре класса, каждый из которых представляет собой реализацию общего для всех классов интерфейса ProjectItem (листинг 2.45). В данном примере этот интерфейс содержит определение метода accept, предназначенного для сохранения ссылки на экземпляр класса Visitor.

Листинг 2.45. ProjectItem.java

```
1. import  java.io.Serializable;
2. import java.util.ArrayList;
3. public interface ProjectItem extends Serializable(
4.   public void accept(ProjectVisitor v);
5.   public ArrayList getProjectItems();
6. }
```

Собственно проект представлен классом Project (листинг 2.48). Класс Deliverable (листинг 2.46) представляет конкретный продукт проекта, а класс Task (листинг 2.49) — его отдельный этап. Кроме того, в листинге 2.47 представлен подкласс класса Task, названный DependentTask. Этот класс содержит набор ссылок на объекты класса Task, влияющие на завершение этапа проекта, который представлен конкретным экземпляром класса DependentTask.

Листинг 2.46. Deliverable.java

```
1. import java.util.ArrayList;
2. public class Deliverable implements ProjectItem{
3.   private String name;
4.   private String description;
5.   private Contact owner;
6.   private double materialsCost;
7.   private double productionCost;
8.
9.   public Deliverable(){ }
10.  public Deliverable(String newName, String newDescription,
11.     Contact newOwner, double newMaterialsCost, double newProductionCost){
12.     name = newName;
13.     description = newDescription;
14.     owner = newOwner;
15.     materialsCost = newMaterialsCost;
16.     productionCost = newProductionCost;
17. }
18.
19.  public String getName(){ return name; }
20.  public String getDescription(){ return description; }
21.  public Contact getOwner(){ return owner; }
22.  public double getMaterialsCost(){ return materialsCost; }
23.  public double getProductionCost(){ return productionCost; }
24.
25.  public void setMaterialsCost(double newCost){ materialsCost = newCost; }
26.  public void setProductionCost(double newCost){ productionCost = newCost; }
27.  public void setName(String newName){ name = newName; }
28.  public void setDescription(String newDescription){ description =
    newDescription; }
29.  public void setOwner(Contact newOwner){ owner = newOwner; }
30.
31.  public void accept(ProjectVisitor v){
32.     v.visitDeliverable(this);
33. }
34.
35.  public ArrayList getProjectItems(){
36.     return null;
37. }
38.}
```

Листинг 2.47. DependentTask.java

```
1. import java.util.ArrayList;
2. public class DependentTask extends Task{
3.   private ArrayList dependentTasks = new ArrayList ();
4.   private double dependencyWeightingFactor;
5.
6.   public DependentTask(){ }
7.   public DependentTask(String newName, Contact newOwner,
8.     double newTimeRequired, double newWeightingFactor){
9.     super(newName, newOwner, newTimeRequired);
10.    dependencyWeightingFactor = newWeightingFactor;
11.  }
12.
13.  public ArrayList getDependentTasks(){ return dependentTasks; }
14.  public double getDependencyWeightingFactor(){ return
    dependencyWeightingFactor; }
15.
```

```
16. public void setDependencyWeightingFactor(double newFactor){
   dependencyWeightingFactor = newFactor; )
17.
18. public void addDependentTask(Task element);
19.    if (!dependentTasks.contains(element)){
20.     dependentTasks.add(element) ;
21.    }
22. }
23.
24. public void removeDependentTask(Task element){
25.    dependentTasks.remove(element);
26. }
27.
28. public void accept(ProjectVisitor v){
29.   v.visitDependentTask(this) ;
30. }
31.}
```

Листинг 2.48. Project.java

```
1. import java.util.ArrayList;
2. public class Project implements ProjectItem{
3.   private String name;
4.   private String description;
5.   private ArrayList projectItems = new ArrayList ();
6.
7.   public Project(){ }
8.   public Project(String newName, String newDescription){
9.     name = newName;
10.    description = newDescription;
11. }
12.
13. public String getName(){ return name; }
14. public String getDescription(){ return description; }
15. public ArrayList getProjectItems(){ return projectItems; }
16.
17. public void setName(Sting newName){ name = newName; }
18. public void setDescription(String newDescription){ description =
   newDescription; }
19.
20. public void addProjectItem(ProjectItem element){
21.    if (!projectItems.contains(element)){
22.      projectItems.add(element);
23.    }
24. }
25.
26. public void removeProjectItem(ProjectItem element){
27.    projectItems.remove(element);
28. >
29.
30. public void accept(ProjectVisitor v){
31.    v.visitProject(this);
32. }
33.}
```

Листинг 2.49. Task.java

```
1. import java.util.ArrayList;
2. public class Task implements ProjectItem{
3.   private String name;
4.   private ArrayList projectItems = new ArrayList ();
5.   private Contact owner;
6.   private double timeRequired;
7.
8.  public Task() { }
9.   public Task(String newName, Contact newOwner,
10.     double newTimeRequired){
11.     name = newName;
12.     owner = newOwner;
13.     timeRequired = newTimeRequired;
14.  }
15.
16.  public String getName(){ return name; }
17.  public ArrayList getProjectItems(){ return projectltems; )
18.  public Contact getOwner(){ return owner; }
19.  public double getTimeRequired(){ return timeRequired; }
20.
21.  public void setName(String newName){ name = newName; }
22.  public void setOwner(Contact newOwner){ owner = newOwner; }
23.  public void setTimeRequired(double newTimeRequired){ timeRequired =
   newTimeRequired; }
24.
25.  public void addProjectItem(ProjectItem element){
26.  if (!projectItems.contains(element)){
27.     projectItems.add(element);
28.  }
29.  }
30.
31.  public void removeProjectItem(ProjectItem element){
32.     projectItems.remove(element);
33.  }
34.
35.  public void accept(ProjectVisitor v){
36.     v.visitTask(this);
37.  }
38.}
```

Базовый интерфейс Projectvisitor, определяющий поведение классов вида Visitor, представлен в листинге 2.50. Для каждого класса проекта интерфейс определяет соответствующий метод `visit`.

Листинг 2.50. ProjectVisitor.java

```
1. public interface ProjectVisitor{
2.   public void visitDependentTask(DependentTask p) ;
3.   public void visitDeliverable(Deliverable p);
4.   public void visitTask(Task p);
5.   public void visitProject(Project p);
6. }
```

Теперь, определив все основные классы, можно определить класс, реализующий интерфейс `ProjectVisitor` и выполняющий вычисления над элементами проекта. В листинге 2.51 представлен такой класс, а именно `ProjectCostVisitor`, предназначенный для расчета стоимости проекта.

Листинг 2.51. ProjectCostVisitor.java

```
1. public class ProjectCostVisitor implements ProjectVisitor{
2.   private double totalCost;
3.   private double hourlyRate;
4.
5.   public double getHourlyRate(){ return hourlyRate; }
6.   public double getTotalCost(){ return totalCost; }
7.
8.   public void setHourlyRate(double rate){ hourlyRate = rate; }
9.
10.  public void resetTotalCost() { totalCost = 0.0; )
11.
12.  public void visitDependentTask(DependentTask p).{
13.     double taskCost = p.getTimeRequired()  * hourlyRate;
14.     taskCost *= p.getDependencyWeightingFactor() ;
15.     totalCost += taskCost;
16.  }
17.  public void visitDeliverable(deliverable p){
18.    totalCost += p.getMaterialsCost()  + p.getProductionCost() ;
19.  }
20.  public void visitTask(Task p){
21.    totalCost += p.getTimeRequired()  * hourlyRate;
22.  }
23.  public void visitProject(Project p)( }
24.}
```

Вся функциональность по выполнению вычислений, равно как и хранение результатов, централизованы в классе `Visitor`. Для того чтобы добавить поддержку новой функциональности, нужно создать соответствующий класс, реализующий интерфейс `ProjectVisitor`, и наполнить нужным содержанием четыре метода `visit` этого класса.

Template Method

Свойства шаблона

Тип: поведенческий шаблон
Уровень: объект

Назначение

Предоставляет метод, который позволяет подклассам перекрывать части метода, не прибегая к их переписыванию.

Представление

Во время работы над проектами часто возникает необходимость оценки издержек на выполнение определенного задания или получение продукта. В PIM-приложении для представления проектов используется много классов. Как минимум, элементы проекта должны быть представлены двумя классами: Task и Deliverable. По мере возрастания сложности проекта, возможно, придется создавать дополнительные классы, такие как Project или DependentTask, чтобы точнее реализовать модель реального проекта.

Конечно, можно в каждом классе создать метод getCostEstimate, однако такой подход ведет к дублированию значительной части программного кода приложения. По мере увеличения количества классов будет все сложнее и сложнее обеспечивать сопровождение кода, распределенного по всем классам проекта.

Более удобный подход состоит в объединении всех взаимосвязанных классов проекта в суперкласс, в котором определен метод getCostEstimate. Однако что делать в тех случаях, когда часть метода getCostEstimate зависит от информации каждого конкретного экземпляра класса Project? Например, как быть, если для класса Task человеко-часы подсчитываются не так, как для класса Deliverable?

В таких ситуациях необходимо определить метод getCostEstimate таким образом, чтобы он вызывал абстрактный метод getTimeRequired, который должен реализовываться классами Task и Deliverable в соответствии с их спецификой. Данный подход, представленный шаблоном Template Method, позволяет получить преимущество от повторного использования кода и при этом не препятствует классам модифицировать определяемое суперклассом поведение в соответствии с их потребностями.

Область применения

Шаблон Template Method рекомендуется использовать в следующих случаях.

- Требуется получить базовый метод, позволив переопределять его части конкретным подклассам.
- Необходимо централизовать функциональность метода, которая остается единой для всех подклассов, но в каждом подклассе она может выполняться по-своему.
- Нужно управлять операциями, которые могут быть перекрыты подклассами.

Описание

При разработке приложения со сложной иерархией классов нередко происходит дублирование одного и того же программного кода в разных частях приложения. Это, конечно же, нежелательно, поэтому нужно прибегать к повторному использованию кода везде, где только можно. Изменение архитектуры классов таким образом, чтобы общие методы были вынесены в суперкласс — это, безусловно, шаг в правильном направлении. Однако проблема состоит в том, что иногда при выполнении вынесенной в суперкласс операции используется информация, доступная лишь на уровне подкласса. Именно поэтому разработчики часто предпочитают отказываться от применения суперклассов, дублируя одинаковый программный код в нескольких подклассах.

В тех случаях, когда многие методы взаимосвязанных классов имеют подобную структуру, может оказать пользу шаблон Template Method. Для его применения нужно сначала определить, какие части методов дублируют друг друга. Затем дублирующуюся функциональность нужно вынести в суперкласс, оставив в подклассах только те методы, которые уникальны для них.

Полученный базовый метод представляет собой структуру операции. Для каждой части операции, которая может изменяться в подклассе, в суперклассе определяется абстрактный метод. Для того чтобы реализовать свою функциональность, подклассы перекрывают такие абстрактные методы. Когда в подклассе вызывается базовый метод, выполняется код, определенный на уровне суперкласса.

Если базовый метод суперкласса определен как `final`, возможности подкласса по перекрытию отдельных частей операции ограничиваются.

Данный шаблон называется Template Method, так как он предоставляет в распоряжение разработчика лишь базовый (template) метод, содержащий структуру операции, в которой некоторые ее этапы представлены вызовами абстрактных методов. Подклассы должны "заполнить пробелы" метода, обеспечив реализацию абстрактных методов на своем уровне.

Реализация

Диаграмма классов шаблона Template Method представлена на рис. 2.16.

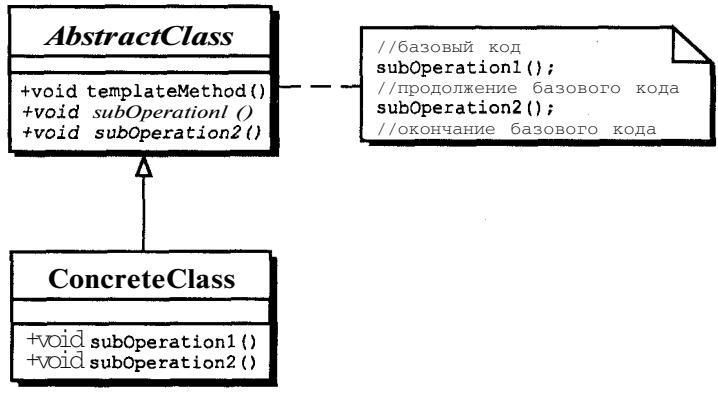


Рис. 2.16. Диаграмма классов шаблона Template Method

При реализации шаблона Template Method обычно используются следующие классы.

- AbstractClass— это абстрактный класс (что неудивительно), содержащий базовый метод и определяющий один или несколько абстрактных методов. Базовый метод содержит "костяк" кода, в котором используются вызовы одного или нескольких абстрактных методов. Для того чтобы подклассы не могли перекрыть базовый метод, его нужно объявить с использованием ключевого слова `final`.

- ConcreteClass — этот класс расширяет класс AbstractClass и реализует абстрактные методы, определенные на уровне последнего. На этом задача класса ConcreteClass исчерпывается, так как структура выполняемой операции определяется базовым методом класса AbstractClass.

Достоинства и недостатки

Основное преимущество шаблона Template Method заключается в том, что он способствует повторному использованию кода. Если шаблон Template Method не используется, приходится дублировать практически один и тот же программный код в разных подклассах. Это преимущество делает шаблон Template Method незаменимым в базовых классах. Такой базовый класс может содержать определения множества методов, зависимость которых от конкретной реализации в подклассах сведена к минимуму. Применяя этот шаблон, можно создать "костяк" целого приложения, для использования которого в конкретной предметной области достаточно будет лишь перекрыть несколько методов.

Однако если базовый метод вызывает слишком много абстрактных методов, использовать класс AbstractClass в качестве суперкласса становится неудобно. Поэтому необходимо стремиться к тому, чтобы базовый метод вызывал лишь несколько абстрактных методов, распределяя избыточную функциональность на несколько суперклассов.

Варианты

Одним из вариантов реализации шаблона Template Method является использование в базовом методе вызовов не абстрактных, а конкретных методов. Иными словами, класс AbstractClass обеспечивает реализацию каждого метода, вызываемого базовым методом, которая будет применяться подклассами по умолчанию. Такие методы называются *хук-методами* (hook method).

Действительно, вполне может возникнуть ситуация, в которой разработчику нет необходимости перекрывать все методы класса AbstractClass, вызываемые базовым методом. К тому же, класс AbstractClass вовсе не обязательно должен быть абстрактным.

Создатель конкретной реализации шаблона Template Method должен позаботиться о надлежащем документировании методов, которые необычным способом используются базовыми методами. При стандартной реализации шаблона в таком документировании нет острой необходимости, поскольку видно, какие методы определены как абстрактные, и, следовательно, должны перекрываться в подклассах. Однако если в реализации шаблона имеются хук-методы и это не отражено в документации, идентифицировать их в качестве таковых невозможно.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Factory Method (стр. 42). Базовые методы часто вызывают методы, созданные в соответствии с шаблоном Factory Method, и которые обеспечивают получение новых экземпляров независимо от того, к какому классу они принадлежат.

- Strategy (стр. 130). Шаблон Strategy с помощью композиции позволяет полностью изменить поведение объекта, тогда как шаблон Template Method с помощью наследования позволяет скорректировать его поведение.
- Intercepting Filter [CJ2EEP]. Шаблон Intercepting Filter использует шаблон Template Method для реализации своей стратегии базового фильтра.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом RunPattern, приведен в разделе "Template Method" на стр. 437 Приложения А.

В данном примере для иллюстрации того, как можно реализовать шаблон Template Method, используются классы PIM-приложения.

В качестве абстрактного класса, определяющего базовый метод, выбран класс `ProjectItem` (листинг 2.52). Его метод `getCostEstimate` возвращает общую стоимость элемента проекта, которая вычисляется по следующей формуле:

стоимость = планируемое_время * почасовая_ставка + стоимость_материалов

Почасовая ставка определяется на уровне класса `ProjectItem` (для этого используются переменная rate, а также соответствующие методы `getRate` и `setRate`), но методы `getTimeRequired` и `getMaterialsCost` являются на уровне этого класса абстрактными. Это означает, что данные методы должны перекрываться подклассами, которые тем самым обеспечивают выполнение вычислений с учетом своей специфики.

Листинг 2.52. ProjectItem.java

```
1. import java.io.Serializable;
2. public abstract class Projectltem implements Serializablef
3.  private String name;
4.  private String description;
5.  private double rate;
6.
7.   public ProjectItem(){}
8.  public ProjectItem(String newName, String newDescription, double newRate){
9.     name = newName;
10.    description = newDescription;
11.    rate = newRate;
12. }
13.
14.  public void setName(String newName){ name = newName; }
15.  public void setDescription(String newDescription){ description =
   newDescription;  }
16.  public void setRate(double newRate){ rate = newRate; }
17.
18.  public String getName() { return name; }
19.  public String getDescription(){ return description; }
20.  public final double getCostEstimate(){
21.     return getTimeRequired() * getRate() + getMaterialsCost();
22. }
```

```
23.  public double getRate(){ return rate; }
24.
25.  public String toString(){ return getName(); }
26.
27.  public abstract double getTimeRequired();
28.  public abstract double getMaterialsCost();
29.}
```

Класс Deliverable (листинг 2.53) представляет в приложении некоторый конкретный продукт, получаемый в ходе выполнения проекта. Так как продукт — это осязаемый физический предмет, методы getTimeRequired и getMaterialsCost класса возвращают заданные фиксированные значения.

Листинг 2.53. Deliverable.java

```
1. public class Deliverable extends ProjectItem{
2.   private double materialsCost;
3.   private double productionTime;
4.
5.   public Deliverable!)( }
6.   public Deliverable(String newName, String newDescription,
7.      double newMaterialsCost, double newProductionTime,
8.      double newRate){
9.     super(newName, newDescription, newRate);
10.    materialsCost = newMaterialsCost;
11.    productionTime = newProductionTime;
12. >
13.
14.  public void setMaterialsCost(double newCost)( materialsCost = newCost; }
15.  public void setProductionTime(double newTime){ productionTime = newTime; }
16.
17.  public double getMaterialsCost(){ return materialsCost; }
18.  public double getTimeRequired(){ return productionTime; }
19.}
```

Класс Task (листинг 2.54) представляет в приложении некую работу, которая может состоять из нескольких этапов или сводиться к получению нескольких продуктов. В этой связи метод getTimeRequired класса определяет общее время, требующееся для выполнения данной работы, путем перебора всех относящихся к ней элементов проекта с вызовом метода getTimeRequired каждого из этих элементов. То же самое выполняет и метод getMaterialsCost, с тем лишь отличием, что он вызывает метод getMaterialsCost каждого дочернего объекта.

Листинг 2.54. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Task extends ProjectItem{
4.   private ArrayList projectItems = new ArrayList();
5.   private double taskTimeRequired;
6.
7.   public TaskO { }
8.   public Task(String newName, String newDescription,
9.      double newTaskTimeRequired, double newRate){
10.    super (newName, newDescription, newRate);
```

```
11.     taskTimeRequired = newTaskTimeRequired;
12.  )
13.
14.  public void setTaskTimeRequired(double newTaskTimeRequired){
   taskTimeRequired = newTaskTimeRequired; }
15.  public void addProjectItem(ProjectItem element){
16.     if (!projectItems.contains(element)){
17.      projectItems.add(element) ;
18.     }
19.  }
20.  public void removeProjectItem(ProjectItem element){
21.     projectItems.remove(element);
22.  )
23.
24.  public double getTaskTimeRequired(){ return taskTimeRequired; }
25.  public Iterator getProjectItemIterator(){ return projectItems.iterator(); }
26.  public double getMaterialsCost(){
27.     double totalCost = 0;
28.     Iterator items = getProjectItemIterator();
29.     while (items.hasNext()){
30.       totalCost += ((ProjectItem)items.next()).getMaterialsCost();
31.     }
32.     return totalCost;
33.  }
34.  public double getTimeRequired(){
35.     double totalTime = taskTimeRequired;
36.     Iterator items = getProjectItemIterator();
37.     while (items.hasNext()){
38.       totalTime += ((ProjectItem)items.next()).getTimeRequired();
39.     }
40.     return totalTime;
41.  }
42.}
```

СТРУКТУРНЫЕ ШАБЛОНЫ

Глава 3

Введение

Структурные шаблоны (structural patterns) с одинаковой эффективностью применяются как для разделения, так и для объединения элементов приложения. Способы воздействия структурных шаблонов на приложение могут быть самые разные. Например, шаблон Adapter может обеспечить возможность двум несовместимым системам обмениваться информацией, тогда как шаблон Facade позволяет отобразить упрощенный пользовательский интерфейс, не удаляя ненужных конкретному пользователю элементов управления.

Структурные шаблоны, рассмотренные в данной главе, имеют следующее назначение.

- *Adapter.* Обеспечение взаимодействия двух классов путем преобразования интерфейса одного из них таким образом, чтобы им мог пользоваться другой класс.
- *Bridge.* Разделение сложного компонента на две независимые, но взаимосвязанные иерархические структуры: функциональную абстракцию и внутреннюю реализацию. Это облегчает изменение любого аспекта компонента.
- *Composite.* Предоставление гибкого механизма для создания иерархических древовидных структур произвольной сложности, элементы которых могут свободно взаимодействовать с единым интерфейсом.
- *Decorator.* Предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций.
- *Facade.* Создание упрощенного интерфейса для группы подсистем или сложной подсистемы.
- *Flyweight.* Уменьшение количества объектов системы с многочисленными низкоуровневыми особенностями путем совместного использования подобных объектов.

- *Half-Object Plus Protocol (HOPP).* Предоставление единой сущности, которая размещается в двух или более областях адресного пространства.
- *Proxy.* Представление другого объекта, обусловленное необходимостью обеспечения доступа или повышения скорости либо соображениями безопасности.

Adapter

Также известен как Wrapper

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Обеспечение взаимодействия двух классов путем преобразования интерфейса одного из них таким образом, чтобы им мог пользоваться другой класс.

Представление

Одним из часто упоминаемых преимуществ объектно-ориентированного программирования является возможность повторного использования кода. Так как данные и поведение моделируемой сущности централизованы в виде некоторого класса, можно (во всяком случае, такая возможность существует в принципе) просто перенести этот класс из одного проекта в другой и с минимальными усилиями использовать в новом проекте его функциональность.

К сожалению, разработчики не всегда могут абсолютно точно предвидеть последующие ситуации. Поэтому, не зная наперед, какие требования будут выдвигаться к написанию исходного кода будущих проектов, он далеко не всегда имеет возможность выбрать оптимальную архитектуру класса, обеспечивающую безусловное повторное использование кода.

Представим, что для ускорения работы над PIM-приложением вы решили объединить усилия с одним из своих друзей, проживающим в другом государстве. Он уже работал над подобным проектом и может предоставить в ваше распоряжение уже реализованную на практике и успешно работающую адресную систему. Однако, получив файлы, вы узнаете, что интерфейс этой системы не соответствует разработанным вами интерфейсам. Что еще хуже, в названиях классов и методов вместо слов английского языка используются слова родного языка вашего друга.

Итак, у вас есть выбор из двух малопривлекательных решений.

- Первое решение — переписать новый компонент таким образом, чтобы он реализовывал все требуемые интерфейсы. Переписывание нового компонента нельзя назвать наилучшим способом решения проблемы, поскольку вам придется проделывать эту работу вновь и вновь при каждом поступлении от вашего друга обновленной им версии компонента.

- Второе решение — переписать ваше приложение таким образом, чтобы оно могло работать с новыми интерфейсами, созданными другим разработчиком с использованием языка, отличного от английского. Недостаток этого решения состоит в том, что вам придется проделать огромный объем работы, внося изменения в исходный код целого приложения. Кроме того, после внесения таких изменений вам будет труднее разобраться в собственном программном коде, так как вы не знаете языка, на котором разговаривает ваш друг, — все названия, которые, с точки зрения вашего друга, говорят сами за себя, для вас будут лишь бессмысленным набором букв.

Итак, поскольку ни одно из решений нельзя назвать удовлетворительным, а с компонентом нужно что-то делать, вы решаете, что нуждаетесь в неком подобии переводчика — компоненте, который бы преобразовывал вызовы одного интерфейса в вызовы другого. Такое решение существует и реализуется в соответствии с шаблоном Adapter. Его работа напоминает работу блока питания, который преобразует один тип электрического тока в другой, несовместимый с первым. Используя шаблон Adapter, ваше приложение может по-прежнему работать со своими интерфейсами и при этом задействовать новые компоненты. Если же будет получена новая версия стороннего компонента, нужно будет внести изменения в классы, созданные в соответствии с шаблоном Adapter.

Область применения

Шаблон Adapter рекомендуется применять в следующих случаях.

- В уже имеющуюся среду необходимо подключить объект, который работает с интерфейсом, отличным от интерфейсов, реализованных в этой среде.
- Возникают ситуации, в которых требуется трансляция интерфейсов, поступающих от разных разработчиков.
- Объект должен выполнять роль посредника для целой группы классов, но какой именно класс будет использован во время работы приложения, на этапе разработки точно не известно.

Описание

Иногда возникает необходимость в использовании класса в новом для него окружении, но при этом хотелось бы не заниматься его "подгонкой". В таких случаях можно создать специальный класс, например, Adapter, и возложить на него задачу трансляции вызовов. Для обращения к методам встраиваемого класса чуждая для него среда обращается не к этому классу, а к методам класса Adapter, который преобразует их в вызовы методов встраиваемого класса.

Типичными представителями приложений, в которых может оказаться полезным шаблон Adapter, являются графические, текстовые или мультимедийные редакторы, поддерживающие подключаемые модули (plug-in). Кроме того, к разряду таких приложений можно отнести Web-броузеры, а также любые приложения, поддерживающие многоязычность, и приложения, использующие подключение компонентов во время работы (например, JavaBeans).

Примером реализации шаблона Adapter является так называемый разговорник — небольшая книга с набором фраз, как минимум, на двух языках, представляющих собой вопросы и ответы на них для большинства типичных ситуаций, в которых может оказаться человек, попавший в другую страну. Используя этот разговорник, он может, совершенно не владея иностранным языком, задать вопрос и получить на него более или менее вразумительный ответ, общаясь с иностранцем, который не знает ни одного слова на его родном языке.

Реализация

Диаграмма классов шаблона Adapter представлена на рис. 3.1.

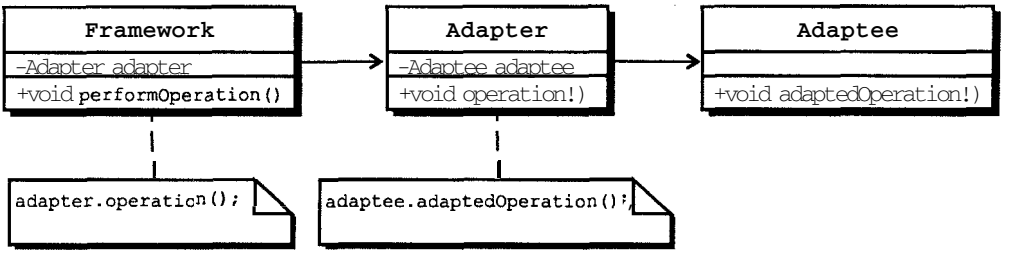


Рис. 3.1. Диаграмма классов шаблона Adapter

При реализации шаблона Adapter обычно используются следующие классы.

- Framework — этот класс использует интерфейс Adapter и либо создает новый объект класса ConcreteAdapter, либо получает на него ссылку из какого-либо источника.
- Adapter — интерфейс, определяющий методы, которые используются классом Framework.
- ConcreteAdapter— реализация интерфейса Adapter. Этот класс хранит ссылку на экземпляр Adaptee и транслирует вызовы методов, поступающие от экземпляра класса `Framework`, в вызовы методов экземпляра класса Adaptee. При трансляции может выполняться преобразование (wrapping), или модификация, параметров и возвращаемых значений.
- Adaptee — интерфейс, в котором определены методы, подлежащие адаптации. Этот интерфейс позволяет конкретной реализации адаптируемого класса динамически подгружаться во время выполнения приложения.
- ConcreteAdaptee — реализация интерфейса Adaptee. Класс, который должен быть адаптирован для того, чтобы его мог использовать класс Framework.

В тех случаях, когда взаимодействие нового компонента со средой носит сложный характер, рекомендуется составлять схему обработки вызовов, чтобы лучше понять, как управлять взаимодействием классов приложения. Схема обработки вызовов — это таблица, в которой показано, как объект-адаптер транслирует вызовы методов и согласует параметры при взаимодействии рабочей среды и адаптируемого объекта. В табл. 3.1 приведен пример схемы обработки вызовов для одного объекта.

Таблица 3.1. Пример схемы обработки вызовов

Рабочая среда	*Действия адаптера*	*Адаптируемый объект*
method1	нет	method2
argument1	нет	argument2
argument2	преобразование типа	argument2
	создание	argument3

Лучше всего подобные схемы представлять в среде, использующей универсальный язык моделирования (UML — Unified Modeling Language), например, в виде последовательной диаграммы (sequence diagram). Пример такой диаграммы представлен на рис.3.2.

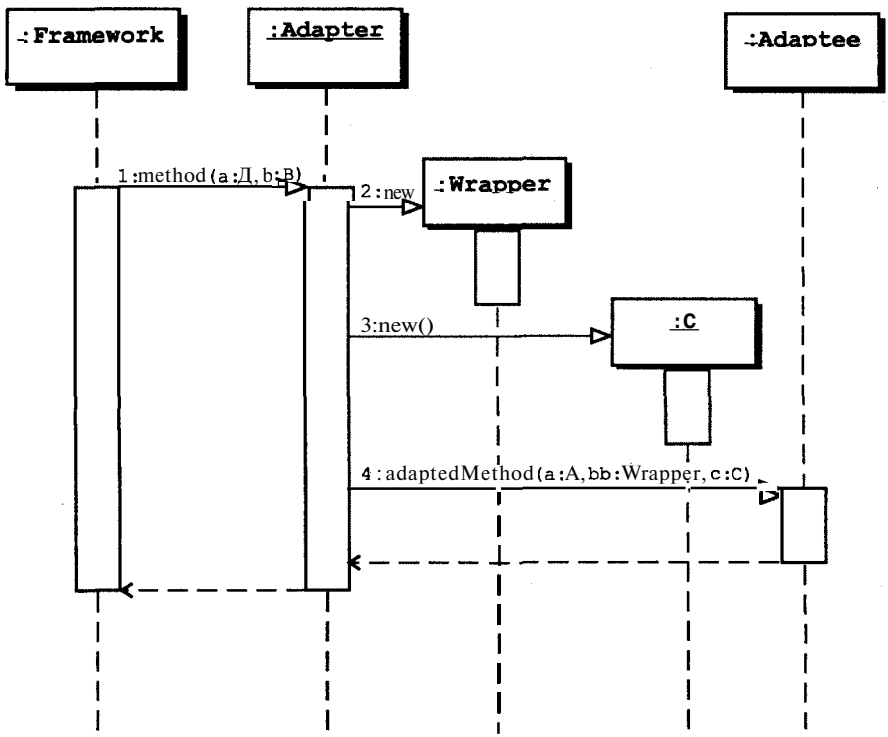


Рис. *3.2.* Диаграмма, представляющая схему обработки вызовов

Достоинства и недостатки

Шаблон Adapter значительно повышает степень повторного использования программного кода, позволяя взаимодействовать двум или более объектам, которые без его помощи были бы несовместимыми. Однако для того чтобы рабочую среду приложения можно было бы без особых проблем адаптировать для работы с классами сторонних разработчиков, необходимо с самого начала как следует продумать архитектуру такой среды. Это требует рассмотрения двух аспектов: функциональной структуры вызовов и трансляции параметров.

Если имеется функциональное несоответствие между вызовами, поступающими из рабочей среды, и адаптируемым объектом, объект-адаптер должен обеспечить, чтобы выполнялись все требования для вызовов методов адаптируемого объекта, в том числе он должен вызвать, если это нужно, дополнительные методы настройки, прежде чем поступит вызов из рабочей среды.

Вторая проблема, которую должен решать объект-адаптер, состоит в корректной трансляции параметров, так как далеко не всегда параметры, передаваемые из рабочей среды, совместимы с параметрами, с которыми работает адаптируемый объект. В таких случаях объект-адаптер обычно либо создает нужные объекты, если между параметрами среды и вызываемого объекта нет прямого соответствия, либо преобразует объект таким образом, чтобы с ним мог работать адаптируемый объект.

Большинство сред общего назначения, в которых используется шаблон Adapter, обычно строится вокруг шаблона Command, используя определенные формы обмена сообщениями или получение информации о классах и объектах (introspection), а также динамическую подгрузку классов (reflection) во время выполнения. В самом общем случае шаблон Command может избавить разработчика от необходимости в использовании шаблона Adapter. (См. раздел "Command" на стр. 71.)

Варианты

По самой своей природе объекты-адаптеры динамичны, поэтому среди них вряд ли можно найти два абсолютно одинаковых. Тем не менее, можно отметить несколько общих вариантов. Ниже изложены особенности трех из множества возможных вариантов.

- *Один адаптер для нескольких адаптируемых объектов.* В некоторых случаях удобно создать такую архитектуру системы, в которой объект-адаптер являлся бы частью рабочей среды. Такой адаптер часто работает как посредник между системой и несколькими адаптируемыми объектами.
- *Адаптеры, не базирующиеся на интерфейсе.* Использование интерфейсов языка Java позволяет создавать очень гибкие адаптеры. Однако бывают ситуации, в которых применения интерфейсов невозможно. Например, интерфейсы оказываются бесполезными, когда нужно адаптировать готовый компонент, который при разработке не реализовывал ни одного интерфейса. В таких случаях можно воспользоваться шаблоном Adapter, не использующим интерфейсов. Конечно, за такое решение приходится расплачиваться потерей гибкости полученной системы.

- *Два интерфейса:* один между вызывающей средой и адаптером, а второй — между адаптером и адаптируемым объектом. Дополнительный промежуточный слой, образуемый интерфейсом, который помещают между вызывающей средой и адаптером, облегчает добавление новых адаптеров во время работы системы. В то же время интерфейс, находящийся между объектом-адаптером и адаптируемым объектом, обеспечивает возможность динамической подгрузки последнего во время работы приложения. Наличие двух таких интерфейсов делает возможной разработку полноценной архитектуры системы с подключаемыми модулями, в которой адаптируемые объекты без каких-либо проблем могут подключаться к системе "на ходу".

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Bridge (стр. 164). Хотя шаблоны Adapter и Bridge очень похожи, они имеют разное назначение. Шаблон Bridge разделяет абстракцию компонента от его реализации, тем самым позволяя изменять любую составляющую независимо от другой. Шаблон Adapter позволяет лишь использовать готовый объект стороннего разработчика, несовместимый с разрабатываемой системой.
- Decorator (стр. 180). Шаблон Adapter предназначен для изменения интерфейса объекта с сохранением его функциональности. Шаблон Decorator, наоборот, оставляет интерфейс объекта неизменным, но при этом меняет его функциональность.
- Proxy (стр. 209). И шаблон Adapter, и шаблон Proxy предоставляют интерфейс с объектом. Различие между ними состоит в том, что интерфейс, предоставляемый шаблоном Adapter, отличается от интерфейса объекта, тогда как интерфейс, предоставляемый шаблоном Proxy, совпадает с ним.
- Business Delegate [CJ2EEP]. Шаблон Business Delegate может использоваться для тех же целей, что и шаблон Proxy. Иными словами, объект, созданный в соответствии с шаблоном Business Delegate, может на локальном уровне представлять уровень бизнес-модели. Кроме того, такой объект может работать и как объект-адаптер, обеспечивая взаимодействие несовместимых систем.

Пример

В данном примере PIM-приложение использует программный интерфейс API, полученный из некоторого внешнего источника, который применяет в разработке программных компонентов язык, отличный от английского. Интерфейс API образуется двумя файлами, которые представляют собой набор готовых покупных классов, предназначенных для представления списка контактных лиц. Базовые операции определены в интерфейсе Chovnatlh (листинг 3.1).

Листинг 3.1. Chovnatlh.java

```
1. publi'c interface Chovnatlh{
2.   public String tlhapWa$DIchPong();
3.   public String tlhapQavPong();
4.   public Strind tlhapPatlh();
5.   public String tlhapGhom();
6.
7.   public void cherWa$DIchPong(String chu$wa$DIchPong);
8.   public void cherQavPong(String chuSQavPong);
9.   public void cherPatlh(String chuSpatlh);
10.  public void cherGhom(String chuSghom);
11.}
```

Реализация этих методов выполнена в тексте класса `ChovnatlhImpl` (листинг 3.2).

Листинг 3.2. ChovnatlhImpl.java

```
1. //pong = name
2. //wa'DIch = first
3. //Qav = last
4. //patlh = rank (title)
5. //ghom = group (organization)
6. //tlhap = take (get)
7. //cher = set up (set)
8. //chu' = new
9. //chovnatlh = specimen (contact)
10.
11.public class Chovnatlhlmpl implements Chovnatlh{
12.  private String waSDIchPong;
13.  private String QavPong;
14.  private String patlh;
15.  private String ghom;
16.
17.  public ChovnatlhImpl(){ }
18.  public ChovnatlhImpl(String chu$wa$DIchPong, String chuSQavPong,
19.     String chu$patlh, String chu$ghom){
20.       waSDIchPong = chu$waSDIchPong;
21.       QavPong = chuSQavPong;
22.       patlh = chuSpatlh;
23.       ghom = chuSghom;
24. }
25.
26.  public String tlhapWa$DIchPong(){ return waSDIchPong; }
27.  public String tlhapQavPong(){ return QavPong; }
28.  public String tlhapPatlh(){ return patlh; }
29.  public String tlhapGhom(){ return ghom; }
30.
31.  public void cherWa$DIchPong(String chuSwaSDIchPong){ waSDIchPong =
   chu$waSDIchPong; }
32.  public void cherQavPong(String chuSQavPong){ QavPong = chuSQavPong; }
33.  public void cherPatlh(String chuSpatlh){ patlh = chu$patlh; }
34.  public void cherGhom(String chuSghom){ ghom = chuSghom; }
35.
36.  public String toString(){
37.    return waSDIchPong + " " + QavPong + ": " +patlh + ", " + ghom;
38. }
39.}
```

Для того чтобы привести в соответствие методы класса ChovnatlhImpl с методами класса Contact (листинг 3.3), нужно воспользоваться объектом-адаптером. Эту задачу решает класс ContactAdapter (листинг 3.4), внутренняя переменная которого содержит ссылку на объект класса Chovnatlhlmpl. Объект класса ContactAdapter управляет данными, образующими информацию о контактном лице: имя, должность и организация, которую это лицо представляет.

Листинг 3.3. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг 3.4. ContactAdapter.java

```
1. public class ContactAdapter implements Contact{
2.   private Chovnatlh contact;
3.
4.   public ContactAdapter(){
5.     contact = new ChovnatlhImpl();
6.   }
7.   public ContactAdapter(chovnatlh newContact){
8.     contact = newContact;
9.   }
10.
11.  public String getFirstName(){
12.     return contact.tlhapWa$DIchPong();
13.  }
14.  public String getLastName(){
15.     return contact.tlhapQavPong();
16.  }
17.  public String getTitleO)
18.     return contact.tlhapPatlh();
19.  }
20.  public String getOrganization(){
21.     return contact.tlhapGhom();
22.  }
23.
24.  public void setContact(Chovnatlh newContact){
25.     contact = newContact;
26.  }
27.  public void setFirstName(String newFirstName){
28.     contact.cherWa$DIchPong(newFirstName);
29.  }
30.  public void setLastName(String newLastName(String newLastName)}
31.     contact.cherQavPong(newLastName);
32.  }
```

```
33.  public void setTitle(String newTitle){
34.     contact.cherPatlh(newTitle);
35.  }
36.  public void setOrganization(String newOrganization){
37.     contact.cherGhom(newOrganization);
38.  }
39.
40.  public String toString(){
41.     return contact.toString();
42.  }
43.}
```

Bridge

Также известен как Handle/Body

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Разделение сложного компонента на две независимые, но взаимосвязанные иерархические структуры: функциональную абстракцию и внутреннюю реализацию. Это облегчает изменение любого аспекта компонента.

Представление

При разработке компонента, представляющего в PIM-приложении список неотложных дел, возникает вполне объяснимое желание обеспечить максимальную гибкость представления этого компонента пользователю, чтобы последний мог использовать маркированные и нумерованные списки или, скажем, списки, элементы которых помечены иероглифами. Кроме того, хотелось бы иметь возможность модификации базовой функциональности списка, что позволит предоставить пользователю выбор между несортированным, упорядоченным и выстроенным по приоритетам списками.

Для обеспечения наличия в программе всех этих возможностей необходимо разработать группу классов списков, каждый из которых обеспечивает определенный тип вывода содержимого списка и организацию находящейся в нем информации. Однако такой подход быстро продемонстрирует свою непрактичность, так как количество комбинаций, учитывающих все варианты вывода содержимого списка на экран И организации списка, растет слишком быстро.

Гораздо удобнее разделить средства представления списка неотложных дел от средств его нижележащей реализации. Эту задачу решает шаблон Bridge путем определения двух классов или интерфейсов, предназначенных для совместной работы. В случае с PIM-приложением такие классы или интерфейсы могут называться, например, `List` и `ListImpl`. Интерфейс `List` представляет функциональность вывода на

экран, а все средства хранения элементов делегирует своей низкоуровневой реализации, т.е. классу `ListImpl`.

Достоинство этого подхода проявляется при добавлении поддержки новых моделей поведения к уже имеющимся. Для добавления буквенных или цифровых обозначений элементов списка достаточно создать подкласс класса List, а для поддержки таких функций, как группирование последовательных элементов, — расширить класс ListImpl. Привлекательность этого решения состоит в том, что разработчик может по своему усмотрению распределять функциональность по двум классам, что позволяет получить суммарную функциональность с более широкими возможностями.

Область применения

Шаблон Bridge рекомендуется применять в следующих случаях.

- Вместо статической связи между абстракцией компонента и его реализацией, необходимо создать гибкую связь.
- Любые изменения в реализации должны быть незаметны для клиентов.
- Идентифицированы абстракции и реализации нескольких компонентов.
- Можно использовать подклассы, но важнее всего обеспечить раздельное управление двумя аспектами системы.

Описание

Иногда необходимо иметь в системе несколько достаточно сложных элементов, внешняя функциональность и нижележащая реализация которых у тех или иных элементов была бы близкой с некоторыми небольшими различиями. В таких случаях механизм наследования не является наилучшим решением, поскольку количество классов, которые нужно создать с его помощью, возрастает, как функция, зависящая сразу от обоих указанных факторов. Например, если нужно реализовать два способа представления и два способа реализации, это приведет к необходимости разработки четырех отдельных классов, а если количество способов представления и реализации увеличится всего лишь на единицу, количество классов возрастет более чем в два раза и составит девять классов (табл. 3.2).

Кроме того, наследование привязывает компонент к статической модели, усложняя его изменение в будущем. Особенно сложно модифицировать готовый компонент, когда требования к нему изменились только после завершения работы над приложением, уже в ходе его эксплуатации. Без сомнения, лучше всего иметь в своем распоряжении какой-то способ, обеспечивающий динамическое изменение обоих аспектов компонента по мере того, как в этом возникает необходимость.

Рассмотрим шаблон Bridge. Этот шаблон решает описанную проблему путем разделения двух указанных аспектов компонента. Получив две разные ветви наследования (одна — по линии функциональности, вторая — по линии реализации), гораздо легче динамически создавать элементы с заданными характеристиками, комбинируя функциональность и реализацию. Таким образом можно добиться высокой гибкости при сравнительно небольших затратах на разработку программного кода.

Наконец, преимущества шаблона Bridge относительно объема дополнительного программного кода особенно заметны по мере увеличения количества вариантов, подлежащих реализации. В табл. 3.2 приведены некоторые цифры, демонстрирующие общее количество классов, которые необходимы для реализации указанного в первых столбцах количества вариантов, при использовании как обычного механизма наследования, так и шаблона Bridge.

Таблица 3.2. Количество подлежащих разработке классов

Внешние представления	*Реализации*	*Классы (наследование)*	*Классы (шаблон Bridge)*
2	2	4	4
3	2	6	5
3	3	9	6
4	3	12	7
4	4	16	8
5	4	20	9

Сравнение наследования и шаблона Bridge

Архитектура шаблона Bridge позволяет выполнять *мультиплексирование* разных вариантов внешнего представления и внутренней реализации компонента. Термин "мультиплексирование" в контексте данного раздела означает возможность ассоциации в любой комбинации внешних и внутренних элементов, что обеспечивает увеличение диапазона возможных вариаций компонента.

Разделение компонента на две отдельные концепции, кроме того, способствует облегчению понимания назначения компонента и его сопровождения. Это объясняется тем, что каждая ветвь наследования выстраивается на основании одной концепции — либо абстракции, либо реализации.

Шаблон Bridge полезен в любой системе, которая должна демонстрировать локальную гибкость во время выполнения. Примером таких систем являются системы с графическим пользовательским интерфейсом, которые должны быть переносимыми на другие платформы. Это требует, чтобы нижележащая реализация применялась только после запуска приложения в конкретной операционной системе. Кроме того, хорошими кандидатами на применение шаблона Bridge являются также приложения, которые изменяют представление своих данных в зависимости от региона (например, представление даты, языка, формата национальной валюты). Наконец, шаблон Bridge часто оказывается эффективным для бизнес-сущностей, которые могут связываться с несколькими разными источниками баз данных.

Концептуальным примером шаблона Bridge является коммутатор службы технической поддержки. Определенное количество линий с определенными номерами позволяют пользователю связаться с меняющимся по количественному и качественному составу техническим персоналом. Понятно, что качество ответа на вопрос пользователя очень сильно зависит от опыта представителя службы технической поддержки.

Кроме того, оно, по-видимому, зависит и от самого вопроса — всегда найдутся пользователи, жалующиеся на сломавшуюся подставку для кофе, которая так удобно выезжала из системного блока при легком нажатии на находящуюся поблизости кнопку.

Реализация

Диаграмма классов шаблона Bridge представлена на рис. 3.3.

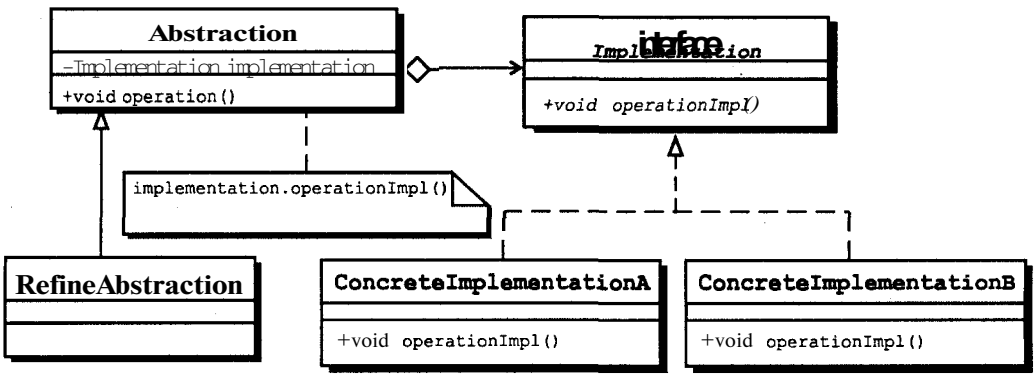


Рис. *3.3.* Диаграмма классов шаблона Bridge

При реализации шаблона Bridge обычно используются следующие классы.

- Abstraction— это класс, который определяет в шаблоне Bridge функциональную абстракцию, формируя тем самым стандартное поведение и структуру. Он содержит ссылку на экземпляр класса Implementation. Этот экземпляр обычно создается с помощью специального метода (чтобы позволить модификацию во время выполнения программы) либо с помощью конструктора.
- `RefineAbstraction` — этот класс расширяет класс Abstraction и формирует дополнительное или модифицированное поведение.
- Implementation— этот интерфейс представляет нижележащую функциональность, используемую экземплярами класса Abstraction.
- `ConcreteImplementation` — этот класс реализует интерфейс Implementation. Он обеспечивает функциональное наполнение поведения и структуры классов `Implementation`.

Достоинства и недостатки

Шаблон Bridge позволяет нескольким объектам абстракций совместно использовать одни и те же объекты нижележащей реализации. Он обеспечивает повышенную гибкость при изменении реализации, причем изменения могут происходить без какого-либо вмешательства со стороны клиента.

При разработке архитектуры приложения, использующего шаблон Bridge, важно четко определить, что относится к функциональной абстракции, а что — к внутренней реализации. Не менее важно решить, каким образом будет представлена истинно

базовая модель реализации шаблона Bridge. Проблема, которая часто возникает при использовании шаблона Bridge, состоит в том, что нередко разработка реализации шаблона выполняется на основе одной или двух возможных вариаций. Опасность заключается в том, что при последующем развитии шаблона выясняется, что некоторые из элементов, считавшихся базовыми, на самом деле представляют собой конкретные вариации, базирующиеся на абстракции и (или) реализации.

Как и в случае со многими другими распределенными объектными шаблонами, возможно, придется подумать, что означает концепция равенства сущностей применительно к шаблону Bridge. Нужно ли сравнивать только абстракции либо только реализации объектов, или же нужно рассматривать их в совокупности?

Варианты

Шаблон Bridge может реализовываться в следующих вариантах.

- *Автоматическое управление.* Некоторые реализации шаблона Bridge созданы таким образом, что могут варьировать своей реализацией без какого-либо вмешательства конечного пользователя. Такое управление осуществляется на основе информации, полученной от приложения или операционной системы.
- *Совместное использование реализации.* Иногда классы, реализующие шаблон Bridge, особенно те, которые не привязываются к конкретному состоянию (т.е. классы, не сохраняющие информацию о внутреннем состоянии), могут использоваться совместно несколькими объектами приложения. Если совместное использование таких классов зависимости применяется довольно широко, такие классы могут выполняться в виде интерфейсов.
- *Единая реализация.* В некоторых случаях создается один класс реализации, который обслуживает несколько классов абстракции. Если применяется единая реализация, то, очевидно, необходимость в определении базового класса для ветви реализации шаблона Bridge отпадает.

Родственные шаблоны

К родственным относятся следующие шаблоны.

- Adapter (стр. 156). Шаблоны Bridge и Adapter очень похожи по структуре, но различны по назначению. Шаблон Bridge разделяет абстракцию компонента от его реализации, тем самым позволяя изменять любую составляющую независимо от другой. Поэтому решение о применении шаблона Bridge должно приниматься до начала разработки приложения, т.е. на этапе проектирования его архитектуры. Шаблон Adapter позволяет использовать готовый объект стороннего разработчика, несовместимый с разрабатываемой системой.
- Singleton (стр. 54). Как отмечалось в предыдущем подразделе, шаблон Singleton может использоваться в тех случаях, когда организовано совместное использование классов реализации.
- Flyweight (стр. 196). Когда древовидная структура становится слишком объемной, применение шаблона Flyweight позволяет уменьшить количество объектов дерева.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `Run Pattern`, *приведен в разделе "Bridge" на стр.* **445** *Приложения* **A.**

В данном примере показано, как использовать шаблон Bridge для расширения функциональности списка неотложных дел PIM-приложения. Сам по себе этот список очень прост и представляет собой обычный объект с возможностью добавления и удаления элементов типа String.

В соответствии с шаблоном Bridge элемент разделяется на две части: абстракцию и реализацию. Реализация — это класс, который выполняет всю работу (в данном примере, он сохраняет элементы в списке и обеспечивает их получение). Общее поведение списка PIM-приложения определяется интерфейсом `ListImpl` (листинг 3.5).

Листинг 3.5. ListImpl.java

```
1. public interface ListImpl{
2.   public void addItem(String item);
3.   public void addItem(String item, int position);
4.   public void removeItem(String item);
5.   public int getNumberOfItems();
6.   public String getItem(int index);
7.   public boolean supportsOrdering();
8. )
```

Класс `OrderedListImpl`, представленный в листинге 3.6, реализует интерфейс ListImpl и сохраняет элементы списка во внутреннем объекте ArrayList.

Листинг 3.6. OrderedListImpl.java

```
1. import java.util.ArrayList;
2. public class OrderedListImpl implements ListImpl{
3.   private ArrayList items = new ArrayList();
4.
5.   public void addItem(String item)(
6.     if (!items.contains(item){
7.       items.add(item);
8.     )
9.   }
10.  public void addItem(String item, int position){
11.    if (!items.contains(item)){
12.      items.add(position, item);
13.    }
14. }
15.
16.  public void removeItem(String item)(
17.    if (items.contains(item)){
18.      items.remove(items.indexOf(item)) ;
19.    )
20. )
```

```
21.
22.  public boolean supportsOrdering(){
23.     return true;
24.  }
25.
26.  public int getNumberOfItems(){
27.     return items.size();
28.  }
29.
30.  public String getItem(int index){
31.     if (index < items.size()){
32.       return (String)items.get(index);
33.     }
34.     return null;
35.  }
36.}
```

Абстракция представляет собой перечень выполняемых над списком операций, который доступен внешнему миру. Класс BaseList (листинг 3.7) содержит перечень основных возможностей списка.

Листинг 3.7. BaseList.java

```
1. public class BaseList{
2.   protected ListImpl implementor;
3.
4.   public void setImplementor(ListImpl impl){
5.      implementor = impl;
6.   }
7.
8.   public void add(String item){
9.      implementor.addItem(item);
10.  }
11.  public void add(String item, int position){
12.     if (implementor.supportsOrdering(){
13.       implementor.addItem(item, position);
14.     }
15.  }
16.
17.  public void remove(String item){
18.     implementor.removeItem(item);
19.  }
20.
21.  public String get(int index){
22.    return implementor.getItem(index) ;
23.  }
24.
25.  public int count (){
26.     return implementor.getNuberOfItems();
27.  }
28.}
```

Обратите внимание на то, что все операции делегированы переменной implementor, представляющей реализацию списка. Какая бы операция не запрашивалась, она в действительности делегируется базовым классом связанному с ним классу конкретной реализации списка.

Расширить возможности базового класса очень просто: достаточно создать подкласс и добавить в него нужную функциональность. Приведенный в листинге 3.8 класс NumberedList наглядно демонстрирует всю мощь шаблона Bridge — достаточно просто перекрыть метод get, и класс уже может обеспечить работу с нумерованными списками.

Листинг 3.8. NumberedList.java

```
1. public class NumberedList extends BaseList{
2.   public String get(int index){
3.     return (index + 1) + ". " + super.get(index) ;
4.   }
5. }
```

Composite

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Предоставление гибкого механизма для создания иерархических древовидных структур произвольной сложности, элементы которых могут свободно взаимодействовать с единым интерфейсом.

Представление

Допустим, разработчик решает расширить возможности PIM-приложения, предоставив в распоряжение пользователя средства управления сложным проектом. В состав таких средств входят средства определения проекта в виде совокупности групп этапов проекта и связанных с последними подэтапов, а также средства, обеспечивающие ассоциацию результатов проекта с этапами. С точки зрения программиста, наиболее естественный способ решения этой задачи состоит в определении древовидной структуры, в которой от корневого этапа (представляющего собственно проект) ответвляются подпроекты, "подподпроекты" и т.д. Иными словами, нужно определить класс Task, содержащий коллекцию ссылок на экземпляры классов Task и Deliverable. Поскольку классы Task и Deliverable являются лишь разными составляющими одного и того же проекта, достаточно определить для них только один шаблон — класс ProjectItem.

Однако что делать в тех случаях, когда пользователю понадобится выполнить какую-то операцию сразу над всем деревом? Например, руководителю проектов нужно получить оценку временных затрат для выполнения всех этапов и получения всех продуктов одного из своих проектов. Для того чтобы снабдить его средствами для

выполнения этой задачи, разработчик должен написать программный код, который последовательно проходил бы по всем ветвям дерева, вызывая соответствующие методы каждой ветви. Но такой подход подразумевает немалый объем работы — нужно создать специальный программный код для перемещения по дереву, вызова всех методов и накопления результатов. Кроме того, так как в каждой ветви в общем случае могут в произвольном порядке встречаться экземпляры двух разных классов (Task и Deliverable), необходимо при подсчете оценки временных затрат обрабатывать их по-разному. Таким образом, при большом количестве классов или сложной структуре дерева требуемый программный код быстро становится трудноуправляемым.

Конечно же, существует и более эффективный способ решения описанной проблемы. Этот способ заключается в применении шаблона Composite, который, опираясь на такие идеи, как полиморфизм и рекурсия, предоставляет разработчику эффективное, простое и удобное в сопровождении решение.

Для начала нужно определить во всех классах, которые могут предоставлять оценку временных затрат, некий стандартный метод, скажем, `getTimeRequired`. Этот метод нужно определить в интерфейсе `ProjectItem`, а затем реализовать его во всех классах, имеющих тип Projectltem. Понятно, что у класса Deliverable метод getTimeRequired возвращает значение 0, так как результат этапа может либо просто отсутствовать, либо присутствовать, но его существование никак не связано с временными затратами. Что касается класса Task, возвращаемое значение складывается из временных затрат текущего этапа и суммы, полученной с помощью вызова метода getTimeRequired всех дочерних экземпляров класса Task.

Метод getTimeRequired, разработанный в соответствии с шаблоном Composite, автоматически вычисляет планируемые временные затраты для любой части дерева. Достаточно лишь вызвать метод getTimeRequired того этапа (т.е. того экземпляра класса `Task`), по которому нужно получить оценку временных затрат, и программный код метода сам выполнит всю работу по перемещению по дереву и вычислению результатов.

Область применения

Шаблон Composite рекомендуется применять в следующих случаях.

- Имеется компонентная модель древовидной структуры ("целое — часть" или "хранилище — хранимое").
- Структура имеет произвольный уровень сложности и по своей природе динамична.
- Необходимо обеспечить единый подход к работе с компонентами структуры, используя одни и те же операции на всех уровнях иерархии.

Описание

Разработчики, придерживающиеся объектно-ориентированных принципов программирования, часто при разработке компонентов стремятся использовать модель "целое — часть" (whole — part). В соответствии с ней, коллекции идентичных объектов (частей) рассматриваются как единая сущность (целое). Обычно такие структуры

должны быть достаточно гибкими и простыми в использовании. Что касается пользователей, то им должны предоставляться средства модификации структуры в процессе работы приложения, добавляя или удаляя части по своему усмотрению. В то же время лучше всего **скрыть** сложность структуры "за кулисами", предоставив пользователям лишь единый, монолитный продукт.

Для того чтобы шаблон Composite соответствовал этим требованиям, необходимо определить такую структуру классов, которая поддерживала бы расширяемость. Такая структура состоит из классов компонента и узла, а также класса, объединяющего их в единое целое.

- *Класс базового компонента* обеспечивает модель сущности, определяя стандартные методы или переменные, которые будут использоваться всеми объектами шаблона Composite.
- *Класс узла* предназначен для реализации функциональности терминальной точки. Иными словами, он представляет такие объекты, которые также входят в единое целое, но сами они не могут содержать других компонентов.
- *Объединяющий класс,* соответствующий всей ветви, обеспечивает добавление новых компонентов, а поэтому и расширяемость своей структуры.

Типовым примером применения на практике шаблона Composite является чертеж, созданный с помощью графического редактора. С отдельными элементами такого чертежа, представляющими собой графические примитивы, можно работать как с единым целым, поскольку они включены в чертеж. Кроме того, один чертеж может содержать в себе как другие чертежи, так и любые комбинации графических примитивов и чертежей любой сложности.

Шаблон Composite может также с успехом использоваться в таких приложениях, как организационные диаграммы, рабочие графики и т.п. Приложения, в которых используются группы, также хорошо подходят для реализации шаблона Composite, конечно, при том условии, что операции группирования можно выполнять рекурсивно и что итоговый продукт, равно как и его элементы, имеет единую функциональность.

Реализация

Диаграмма классов шаблона Composite представлена на рис. 3.4.
Шаблон Composite подразумевает создание трех следующих элементов.

- Component — это интерфейс, определяющий методы, которые должны быть доступными всем частям древовидной структуры. В тех случаях, когда требуется обеспечить стандартное поведение всех подтипов, можно выполнить Component в виде абстрактного класса. Обычно в приложении не создаются экземпляры этого класса, а используются экземпляры его подклассов или реализующие классы, называемые узлами, из которых и образуется древовидная структура.

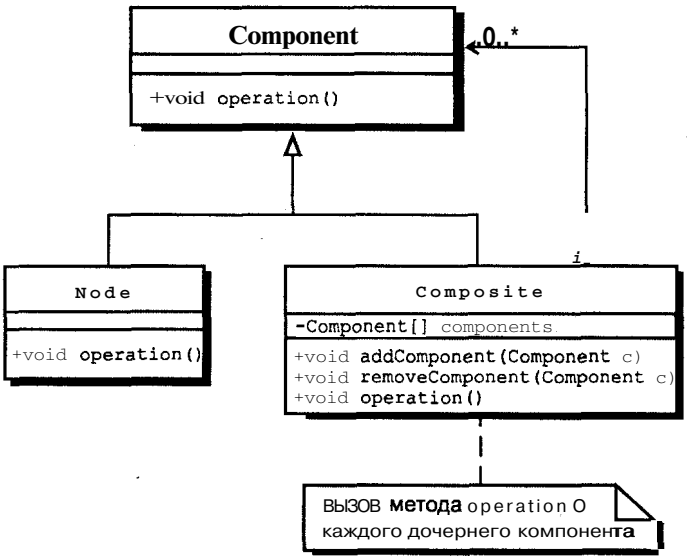


Рис. 3.4. Диаграмма классов шаблона Composite

- Composite — это класс, представляющий собой композицию своих компонентов, т.е. функциональность его определяется содержащимися в нем компонентами. Поскольку класс Composite должен предоставлять возможность получения динамических групп объектов Component, в нем имеются методы добавления экземпляров класса Component в коллекцию и методы их удаления из нее. Методы, определенные в классе Component, реализуются таким образом, чтобы, во-первых, они выполняли функции, характерные для содержащего экземпляры компонентов объекта Composite, а во-вторых, чтобы они вызывали такие же методы для каждого из своих узлов. Такие классы Composite часто называют классами ветвей или контейнеров.
- Node — класс, реализующий интерфейс Component и предоставляющий реализацию для каждого метода Component. Различие между классами Composite и Node состоит в том, что последний не содержит никаких ссылок на другие экземпляры класса Component. Иными словами, совокупность объектов Node представляет собой самый нижний, или пограничный, уровень всей структуры в целом.

При реализации данного шаблона необходимо решить, должен ли каждый компонент иметь ссылку на свой контейнер (т.е. экземпляр класса Composite). Наличие такой ссылки, с одной стороны, позволяет облегчить перемещение по дереву, а с другой — ведет к снижению гибкости приложения.

Достоинства и недостатки

Шаблон Composite дает разработчику возможность использования весьма удачной комбинации: с одной стороны — это достаточно гибкая структура, а с другой — чрезвычайно удобный в управлении интерфейс.

Структура может меняться в любое время путем добавления и удаления объектов класса Component с помощью вызова соответствующих методов экземпляра класса `Composite`. С другой стороны, возможность заменять компоненты ветви означает, что у разработчика имеется возможность изменить и поведение всего класса ветви в целом.

Отдельным достоинством шаблона является то, что в любом месте древовидной структуры нужно всегда вызывать один и тот же метод индивидуальных компонентов.

Использование интерфейсов еще более увеличивает гибкость. Интерфейсы позволяют создавать на основе шаблона целые базовые системы (framework), поддерживающие добавление новых типов во время работы приложения.

С другой стороны, применение интерфейсом может обернуться и другой стороной в тех случаях, когда нужно определить атрибуты и реализовать используемые по умолчанию функции, которые должны быть унаследованы всеми узлами. В таком случае лучше определять Component в виде абстрактного класса.

Еще один недостаток шаблона является следствием его достоинства, а именно высокой гибкости. Нетрудно догадаться, что реализация динамического по своей сути шаблона Composite может быть сопряжена с проблемами при тестировании и отладке. Обычно это выражается в том, что разработчик должен позаботиться о выработке гораздо более серьезной стратегии тестирования и проверки работоспособности приложения, чем в том случае, когда используется традиционная концепция иерархии объектов вида "целое — часть". Если тестирование окажется слишком сложным, лучше всего встроить средства тестирования непосредственно в реализацию класса Composite.

Кроме всего вышесказанного, для успешной реализации шаблона Composite, как правило, требуется либо заранее знать моделируемую структуру (иными словами, иметь детализированную архитектуру класса Composite), либо иметь развитой механизм подгрузки классов. Интерфейсная форма такого шаблона, которая подробнее рассматривается в следующем подразделе "Варианты", может стать полезной альтернативой для обеспечения динамической корректировки поведения системы во время выполнения.

Варианты

Среди различных вариантов реализации шаблона Composite можно выделить следующие типичные решения.

- *Корневой узел.* Для повышения управляемости в системе некоторые разработчики, реализующие шаблон Composite, определяют один объект, отличающийся от остальных объектов и работающий в качестве базы для всей иерархии объектов, выполненной в соответствии с шаблоном Composite. Если такой корневой объект представлен в виде отдельного класса, при его реализации можно использовать шаблон Singleton. Второй подход заключается в предоставлении доступа к корневому объекту, который не имеет ничего общего с шаблоном Singleton, посредством экземпляра класса, выполненного в соответствии с этим шаблоном.
- *Создание ветвей на основе правил.* В структурах высокой сложности (обычно таковыми являются структуры, объединяющие множество разнотипных узлов и ветвей) часто применяется механизм, определяющий, как и при каких условиях узлы могут добавляться к ветвям определенных типов.

Родственные шаблоны

К родственным относятся следующие шаблоны.

- Chain of Responsibility (стр. 62). Использование этого шаблона вместе с шаблоном Composite позволяет "распространять" метод вверх по дереву, т.е. от узлов к ветвям.
- Flyweight (стр. 196). Когда древовидная структура становится слишком объемной, для уменьшения количества избыточных или подобных объектов можно воспользоваться шаблоном Flyweight.
- Iterator (стр. 87). Этот шаблон может применяться совместно с шаблоном Composite для инкапсуляции функциональности перемещения по дереву, если она становится слишком сложной.
- Visitor (стр. 136). При использовании этого шаблона обеспечивается централизация функциональности, которая при использовании лишь одного шаблона Composite должна распределяться по классам ветвей и узлов.
- Composite View [CJ2EEP]. Шаблон Composite описывает, как по отдельным представлениям (view) получить некое единое представление, которое в свою очередь может входить в представление более высокого уровня, что делает его подобным шаблону Composite.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Composite" на стр. 449 Приложения ***А.***

Диаграмма классов, реализующих шаблон Composite в рассматриваемом примере, представлена на рис. 3.5.

В примере показано, как с помощью шаблона Composite можно обеспечить определение времени, необходимого для выполнения всего проекта или его отдельного этапа. Пример включает четыре основных составляющих.

- `Deliverable`. Класс, представляющий конечный продукт, получаемый в результате завершения текущего этапа.
- `Project`. Класс, являющийся корнем композиции и представляющий весь проект в целом.
- `ProjectItem`. Интерфейс, описывающий общую для всех элементов проекта функциональность. Именно в этом интерфейсе содержится объявление метода `getTimeRequired`.
- `Task`. Класс, представляющий коллекцию операций, подлежащих выполнению. Каждый экземпляр этого класса содержит ссылку на коллекцию объектов класса `ProjectItem`.

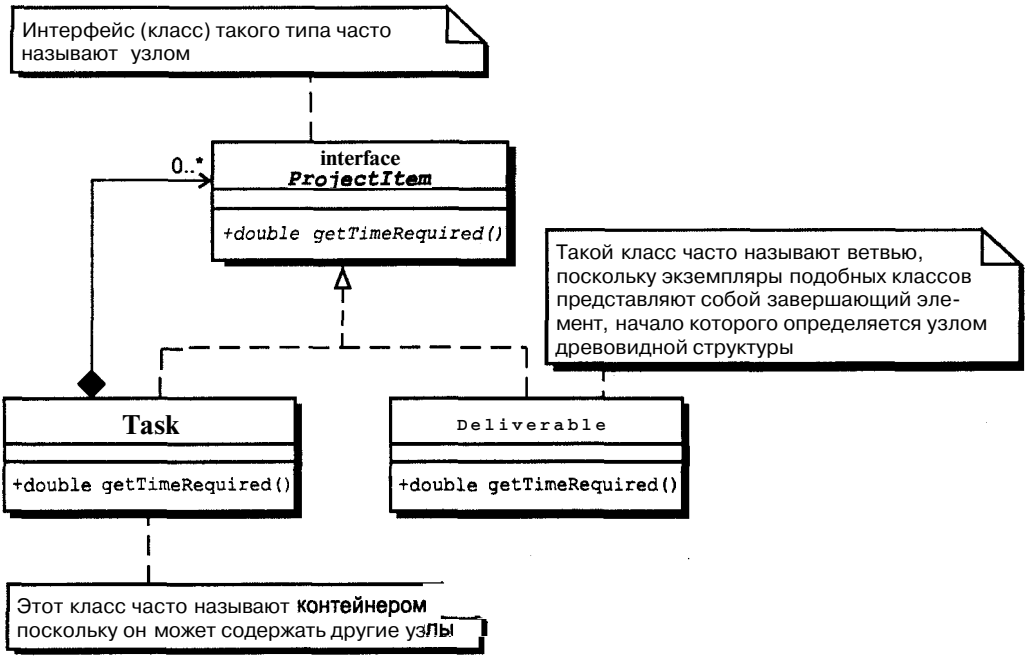


Рис. 3.5. Диаграмма классов примера, реализующих шаблон Composite

Общая функциональность, которой должен обладать любой объект проекта, определяется в интерфейсе Projectltem (**листинг 3.9**). В рассматриваемом примере в этом интерфейсе содержится объявление лишь одного метода getTimeRequired.

Листинг 3.9. ProjectItem.java

```
1. import java.io.Serializable;
2. public interface Projectltem extends Serializable{
3.   public double getTimeRequired();
4. }
```

Поскольку элементы проекта могут образовывать древовидную структуру, классы Deliverable и Task реализуют интерфейс Projectltem. Класс Deliverable (листинг ЗЛО) представляет собой терминальный узел, который не может ссылаться на другие элементы проекта.

Листинг 3.10. Deliverable.java

```
1. public class Deliverable implements ProjectItem{
2.   private String name;
3.   private String description;
4.   private Contact owner;
5.
6.   public Deliverable (){ }
7.   public Deliverable(String newName, String newDescription,
```

```
8.      Contact newOwner){
9.     name = newName;
10.    description = newDescription;
11.    owner = newOwner;
12. }
13.
14. public String getName(){ return name; }
15. public String getDescription(){ return description; }
16. public Contact getOwner(){ return owner; }
17. public double getTimeRequired(){ return 0; }
18.
19. public void setName(String newName){ name = newName; }
20. public void setDescription(String newDescription){ description =
   newDescription; }
21. public void setOwner(Contact newOwner){ owner = newOwner; }
22.}
```

Классы Project (**листинг 3.11**) и Task (**листинг 3.12**), напротив, являются нетерминальными узлами, из-за чего оба класса сохраняют ссылку на коллекцию элементов проекта, представляющих дочерние этапы проекта Task и связанные с ними продукты Deliverable.

Листинг 3.11. Project.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Project implements ProjectItem{
4.   private String name;
5.   private String description;
6.   private ArrayList projectItems = new ArrayList();
7.
8.  public Project (){ }
9.   public Project(String newName, String newDescription){
10.     name = newName;
11.     description = nevDescription;
12. }
13.
14. public String getName(){ return name; )
15. public String getDescription(){ return description; }
16. public ArrayList getProjectItems(){ return projectltems; }
17. public double getTimeRequired(){
18.     double totalTime = 0;
19.     Iterator items = projectItems.iterator ();
20.     while(items.hasNext(){
21.       ProjectItem item = (Projectltem) items .next ();
22.       totalTime += item.getTimeRequired();
23.     }
24.     return totalTime;
25. }
26.
27. public void setName(String newName){ name = newName; }
28. public void setDescription(String newDescription){ description =
   newDescription; }
29.
30.  public void addProjectItem(ProjectItem element){
31.     if (!projectItems.contains(element)){
32.       projectItems.add(element);
33.     }
34. }
```

```
35.  public void removeProjectItem(ProjectItem element){
36.     projectItems.remove(element);
37.  }
38.}
```

Листинг 3.12. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator:
3. public class Task implements Projectltem)
4.  private String name;
5.  private String details;
6.  private ArrayList projectltems = new ArrayListO;
7.  private Contact owner;
8.  private double timeRequired;
9.
10. public Task() { )
11. public Task(String newName, String newDetails,
12.    Contact newOwner, newTimeRequired))
13.    name = newName;
14.    details = newDetails;
15.    owner = newOwner;
16.    timeRequired = newTimeRequired;
17. }
18.
19. public String getName(){ return name; }
20.  public String getDetails(){ return details; }
21.  public ArrayList getProjectItems(){ return projectltems; }
22.  public Contact getOwner(){ return owner; }
23.  public double getTimeRequired(){
24.     double totalTime = timeRequired;
25.     Iterator items = projectItems.iterator();
26.    while(items.hasNext() ){
27.       Projectltem item =  (ProjectItem)items.next();
28.       totalTime += item.getTimeRequired();
29.    }
30.    return totalTime;
31. }
32.
33.  public void setName(String newName)[ name = newName; }
34.  public void setDetails(String newDetails)) details = newDetails; }
35.  public setOwner(Contact newOwner)) owner = newOwner; }
36.  public void setTimeRequired(double newTimeRequired)) timeRequired =
   newTimeRequired; }
37.
38.  public void  addProjectItem(ProjectItem element))
39.     if  (!projectItems.contains(element)){
40.       projectItems.add(element);
41.     }
42.  }
43.  public void removeProjectItem(ProjectItem element))
44.    projectItems.remove(element) ;
45.  }
46.}
```

Лучше всего исследовать работу шаблона Composite, изучая, как работает метод `getTimeRequired`. Для того чтобы получить оценку временных затрат для любой части проекта, достаточно просто вызвать метод getTimeRequired объекта Task (если нужна оценка для всего проекта, следует вызвать аналогичный метод объекта Project). В зависимости от реализации, этот метод возвращает одно из следующих значений.

- Deliverable. Возвращает 0.
- Project или Task. Возвращает значение, представляющее временные затраты, необходимые для выполнения текущего этапа, плюс значение, представляющее собой сумму всех значений, полученных при вызове метода getTimeRequired всех элементов проекта, связанных с текущим элементом.

Decorator

Также известен как Wrapper

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций.

Представление

С помощью рассмотренного в предыдущем разделе шаблона Composite в PIM-приложение можно добавить функциональность, обеспечивающую управление проектами. В соответствии с шаблоном Composite, проект представляется в виде иерархии объектов класса Task и Deliverable, а корневым является объект класса Project. Все классы реализуют интерфейс `ProjectItem`, что идентифицирует их как классы, принадлежащие некоторому проекту.

Но как быть в том случае, если нужно расширить базовые средства классов Task и Deliverable, добавив в них, например, следующую функциональность?

- *Зависимые элементы.* Элементы проекта, завершение которых зависит от экземпляра класса `Task` или Deliverable.
- *Вспомогательные документы.* Объекты класса Task или Deliverable, которые могут ссылаться на дополнительные документы.

Если для добавления данной функциональности воспользоваться механизмом наследования, т.е. создать новые подклассы, это потребует немалого труда по разработке дополнительного программного кода. Например, для того чтобы одни только по-

томки класса Deliverable получили средства поддержки указанных функций, нужно разработать четыре класса: Deliverable, DependentDeliverable, SupportedDeliverable, SupportedDependentDeliverable.

Столкнувшись с этим недостатком, программист, возможно, решит для добавления новой функциональности прибегнуть к композиции. Однако добавление в оба класса Task и Deliverable средств поддержки нужных функций приводит к дублированию одного и того же кода в приложении. Как минимум, это скажется на увеличении объема кода и степени его сложности.

Что если вместо этого сгенерировать классы, обладающие возможностью "подключения на ходу" (plugin). Тогда не придется вносить изменения, призванные обеспечить поддержку новых функций, непосредственно в классы Task и Deliverable — достаточно лишь создать зависимые классы, которые можно было бы подключить с целью расширения функциональности к любому элементу проекта. Можно сравнить это с программным эквивалентом операции, которую мы выполняем, подключая комплект устройств для создания эффекта объемного звука к стандартному стереовыходу. Действительно, возможности оборудования на самом деле не изменились, просто в нашем распоряжении появились некоторые новые средства. Возвращаясь к программированию, предположим, что определены классы DependentProjectItem и SupportedProjectItem. В каждом из этих классов содержится программный код, обеспечивающий поддержку лишь узкого круга задач, связанных с назначением класса, а также ссылка на реальный элемент проекта, который экземпляр данного класса расширяет. Таким образом, резко уменьшается объем программного кода, а разработчик при этом получает гораздо большую свободу в выборе любых комбинаций таких "декоративных" классов (т.е. классов, выполненных в соответствии с шаблоном Decorator) для добавления целых групп определенных свойств к стандартным свойствам элементов проекта.

Область применения

Шаблон Decorator рекомендуется использовать в следующих случаях.

- Необходимо осуществлять динамическое изменение свойств классов, причем незаметно для пользователя и не связываясь с ограничениями, присущими механизму наследования.
- Свойства могут подключаться к компоненту или отключаться от него во время работы системы.
- Имеется несколько независимых функций, которые нужно применять динамически и в любой комбинации.

Описание

Некоторые объекты реального мира имеют сложную функциональность и (или) структуру, расширение или уменьшение которых в точности соответствует компонентной модели. Примером такого объекта может быть карта с прилагаемыми к ней прозрачными пленками, на которых нанесены города или возвышенности — объект остается одним и тем же, но его свойства гибко изменяются в зависимости от текущих требований.

Шаблон Decorator работает по такому же принципу — он обеспечивает добавление дополнительных слоев к базовому объекту, а также в случае необходимости замену одних слоев другими. Каждый слой дополняет базовый объект своим поведением (методами) и состоянием (переменными). Шаблон позволяет объединять и как угодно ассоциировать слои друг с другом и базовым объектом, что дает возможность получить сложные модели поведения объекта из набора относительно простых строительных блоков.

Естественно, шаблон Decorator как нельзя лучше подходит для использования в приложениях, реализующих функциональность динамически создаваемых многослойных видов и представлений. Одним из примеров таких приложений является семейство продуктов для обеспечения групповой работы (groupware), позволяющих участникам рабочей группы по сети редактировать один и тот же базовый документ. Кроме того, этот шаблон может с успехом применяться в графических редакторах, а также в большинстве приложений, связанных с форматированием фрагментов текста, отдельных абзацев или целых документов. На низком уровне шаблон Decorator можно успешно применять для получения заданной функциональности путем применения к базовой модели определенной комбинации фильтров. Среди примеров можно выделить потоковый **ввод/вывод**, а также коммуникационные подключения (сокеты), подобные объекту `BufferReader`, который позволяет считывать строку за строкой из объекта Reader.

Применение шаблона Decorator можно сравнить с тем, как в автосалоне предлагаются различные варианты оборудования автомобиля. Покупатель выбирает базовую модель, а затем решает, каков будет цвет кузова, материал и цвет обивки салона, какое будет установлено оборудование и т.п. При "добавлении" к автомобилю каждого нового "слоя" он приобретает новые характеристики, что, естественно, отражается на цене. (Конечно, отличие от шаблона Decorator имеется, и весьма существенное: покупатель не может с такой же легкостью, как он это делал во время заказа, изменить свой выбор после того, как приобретет автомобиль.)

Реализация

Диаграмма классов шаблона Decorator представлена на рис. 3.6.

При реализации шаблона Decorator обычно используются следующие классы.

- Component. Представляет компонент, реализующий общую модель поведения. Может быть как абстрактным классом, так и интерфейсом.
- Decorator. Абстрактный класс или интерфейс, который определяет стандартное поведение, общее для всех экземпляров ConcreteDecorator. Этот класс обеспечивает поддержку хранения информации: в нем содержится ссылка на Component, который может быть представлен как классом ConcreteComponent, так и классом ConcreteDecorator. Если иерархия классов Decorator создается в виде подклассов тех классов, которые они расширяют, эта ссылка может использоваться для каких-то других целей.
- Один или более классов ConcreteDecorator. Каждый подкласс класса Decorator должен поддерживать возможность образования связных списков (ссылка на компонент плюс средства для добавления и удаления этой ссылки). Помимо базовых требований, каждый класс ConcreteDecorator может определять дополнительные методы и (или) переменные для расширения компонента.

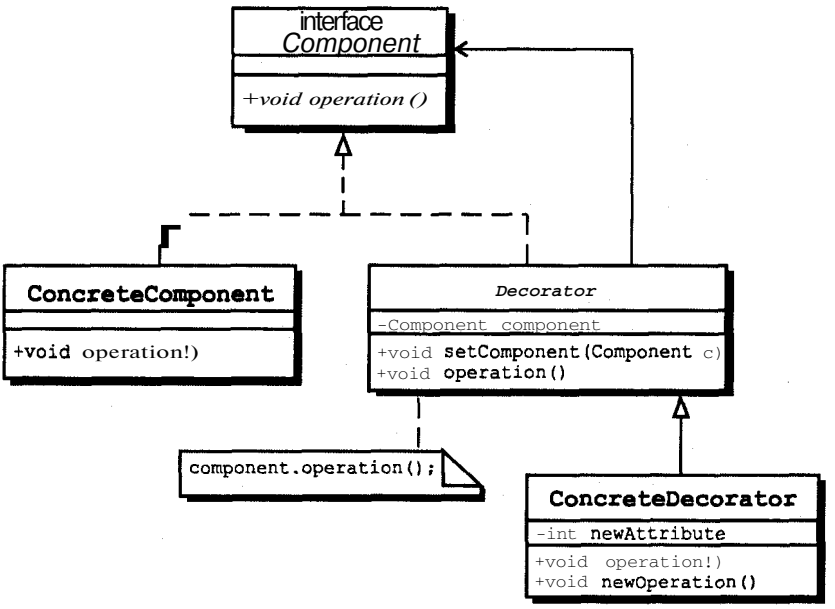


Рис. *3.6.* Диаграмма классов шаблона Decorator

Достоинства и недостатки

Шаблон Decorator предоставляет разработчику возможность без особых усилий настраивать и корректировать поведение объекта во время выполнения программы. Кроме того, он значительно упрощает программирование, так как в соответствии с этим шаблоном разработчик, вместо программирования функциональности в самом компоненте, создает серию классов, каждый из которых отвечает за определенную часть функциональности. Это также позволяет сделать компонент более простым с точки зрения его расширяемости в будущем, поскольку для таких изменений достаточно лишь написать несколько новых классов, не затрагивая уже имеющихся.

В зависимости от поведения некоторые слои, выполненные в соответствии с шаблоном Decorator, могут совместно использоваться несколькими объектами (обычно это слои, поведение которых никак не связано с их состоянием). Это позволяет экономить память, выделяемую для системы.

В самом крайнем варианте реализация шаблона Decorator обычно проявляется в создании большого количества слоев: иными словами, между пользователем и реальным объектом размещается множество маленьких объектов. Такое решение может иметь много неприятных последствий, таких как затруднение отладки и тестирования или снижение производительности системы, вызванное некорректно выбранной архитектурой реализации шаблона.

Равенство объектов в системе всегда должно обрабатываться корректно. Этот принцип особенно важен в отношении шаблона Decorator, поскольку объекты "сидят друг перед другом". Обычно если в приложении требуется выполнять сравнение объектов, необходимо таким образом написать операцию сравнения, чтобы в ней либо

идентифицировался нижележащий объект, либо оценивалась комбинация базового объекта с порядком и "значимостью" каждого слоя.

Наконец, для корректного удаления слоев из системы могут потребоваться дополнительные усилия, поскольку они могут находится в каком угодно месте цепочки. Для того чтобы упростить задачу в некоторых реализациях шаблона Decorator разработчики определяют как ссылку на следующий, так и ссылку на предыдущий объекты, что облегчает процедуру удаления.

Варианты

Шаблон Decorator может реализовываться в следующих вариантах.

- Как отмечалось в предыдущем подразделе "Достоинства и недостатки", в некоторых случаях желательно разрабатывать классы шаблона таким образом, чтобы они хранили ссылки как на следующий, так и на предыдущий объекты, что облегчает их удаление во время работы приложения.
- В некоторых реализациях шаблона Decorator абстрактный класс Decorator не используется. Как правило это происходит в тех случаях, когда у компонента может быть лишь один дополнительный вариант.
- Можно создать перекрывающиеся классы Decorator, которые будут переопределять некоторые части поведенческой модели компонента. Однако при использовании таких классов нужно быть осторожным, поскольку компоненты в данном варианте могут вести себя непредсказуемо (если, конечно, в программном коде нет жестко установленных правил, определяющих, когда и как может перекрываться поведение компонента).

Родственныешаблоны

К родственным можно отнести следующие шаблоны.

- Adapter (стр. 156). Шаблон Adapter предназначен для изменения интерфейса без внесения изменений в функциональность, тогда как шаблон Decorator оставляет неизменным интерфейс, но меняет функциональность.
- Composite (стр. 171). Шаблон Decorator можно рассматривать как облегченную версию шаблона Composite: вместо хранения коллекции компонентов, класс Decorator хранит лишь одну ссылку на другой компонент. Другое отличие состоит в том, что Decorator расширяет функциональность, а не передает вызовы методов.
- Strategy (стр. 130). Шаблон Decorator используется для модификации или расширения внешней функциональности объекта, тогда как шаблон Strategy — для модификации его внутреннего поведения.
- Intercepting Filter [CJ2EEP]. Этот шаблон использует шаблон Decorator для получения модифицированного системного запроса без изменения самого запроса.

Пример

Примечание

*Полный работающий код данного примера со вспомогательными классами, а также классом RunPat tern, приведен в разделе "Decorator" на стр. **456** Приложения **A.***

В данном примере показано, как с помощью шаблона Decorator можно расширить возможности объектов, представляющих элементы проекта. Основой программного представления проекта является интерфейс ProjectItem (**листинг 3.13**). Этот интерфейс должен реализовываться любым классом, который будет использован в проекте. В данном примере интерфейс Projectltem содержит определение лишь одного метода getTimeRequired.

Листинг 3.13. ProjectItem.java

```
1. import java.io.Serializable;
2. public interface Projectltem extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public double getTimeRequired();
5. }
```

Интерфейс Projectltem реализуется классами Deliverable (листинг 3.14) и Task (листинг 3.15), в которых содержится определение базовой функциональности двух типов элементов проекта. Как и в предыдущих примерах, класс Task представляет определенный этап проекта, а класс Deliverable — какой-то конкретный продукт.

Листинг 3.14. Deliverable.java

```
1. public class Deliverable implements ProjectItem{
2.   private String name;
3.   private String description;
4.   private Contact owner;
5.
6.   public Deliverable(){ }
7.   public Deliverable(String newName, String newDescription,
8.     Contact newOwner) {
9.     name = newName;
10.    description = newDescription;
11.    owner = newOwner;
12.  }
13.
14.  public String getName(){ return name; }
15.  public String getDescription(){ return description; }
16.  public Contact getOwner(){ return owner; }
17.  publ:-c double getTimeRequired (){ return 0; }
18.
19.  public void setName(string newName){ name = newName; }
20.  public void setDescription(String newDescription){ description =
   newDescription; } '
21.  public void setOwner(Contact newOwner){ owner = newOwner; }
```

```
22.
23.  public String toString(){
24.     return "Deliverable: " + name;
25.  }
26.}
```

Листинг 3.15. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Task implements ProjectItem{
4.   private String name;
5.   private ArrayList projectItems = new ArrayList ();
6.   private Contract owner;
7.   private double timeRequired;
8.
9.   public Task(){ }
10.  public Task(String newName, Contact newOwner,
11.     double newTimeRequired){
12.     name = newName;
13.     owner = newOwner;
14.     timeRequired = newTimeRequired;
15.     }
16.
17.  public String getName(){ return name; }
18.  public ArrayList getProjectItems(){ return projectItems; }
19.  public Contact getOwner(){ return owner; }
20.  public double getTimeRequired(){
21.     double totalTime = timeRequired;
22.     Iterator items = projectItems.iterator();
23.     while(items.hasNext()){
24.       ProjectItem =  (ProjectItem)items.next();
25.      totalTime += item.getTimeRequired() ;
26.     }
27.     return totalTime;
28.  }
29.
30.  public void setName(String newName){ name = newName; }
31.  public void setOwner(Contact newOwner){ owner = newOwner; }
32.  public void setTimeRequired(double newTimeRequired){ timeRequired =
   newTimeRequired; }
33.
34.   public void addProjectItem(ProjectItem element){
35.     if (!projectItems.contains(element)){
36.       projectItems.add(element) ;
37.     }
38.  }
39.  public void removeProjectItem(ProjectItem element){
40.     projectItems.remove(element);
41.  }
42.
43.  public String toString(){
44.     return "Task: " + name;
45.  ]
46.}
```

Теперь настало время расширить базовые средства этих классов. Класс ProjectDecorator (листинг 3.16) обеспечивает принципиальную возможность дополнения классов Task и Deliverable.

Листинг 3.16. ProjectDecorator.java

```
1. public abstract class ProjectDecorator implements ProjectItem{
2.   private ProjectItem projectItem;
3.
4.   protected ProjectItem getProjectItem(){ return projectItem; }
5.   public void setProjectItem(ProjectItem newProjectItem){ projectItem =
   newProjectItem; }
6.
7.   public double getTimeRequired(){
8.      return projectItem.getTimeRequired();
9.   }
10.}
```

Класс ProjectDecorator реализует интерфейс ProjectItem и содержит переменную, ссылающуюся на следующий экземпляр класса ProjectItem, представляющий "декорируемый" элемент. Необходимо отметить, что класс ProjectDecorator делегирует вызов метода getTimeRequired своему внутреннему элементу. Это можно проделать и с любым другим методом, который зависит от функциональности нижележащего компонента. Действительно, если на выполнение этапа проекта отводится пять дней, метод должен вернуть значение, соответствующие пяти дням, вне зависимости от того, какие дополнительные возможности были добавлены к классу Task путем "декорирования".

В данном примере приведено два подкласса класса ProjectDecorator, демонстрирующих, как можно добавить дополнительную функциональность к элементам проекта. Класс DependentProjectItem (листинг 3.17) использован для того, чтобы показать, что объект класса Task или Deliverable зависит от завершения какого-то другого элемента проекта, представленного объектом класса ProjectItem.

Листинг 3.17. DependentProjectItem.java

```
1. public class DependentProjectItem extends ProjectDecorator{
2.   private ProjectItem dependentItem;
3.
4.   public DependentProjectItem(){ }
5.   public DependentProjectItem(ProjectItem newDependentItem){
6.      dependentItem = newDepentItem;
7.   }
8.
9.   public ProjectItem getDependentItem(){ return dependentItem; }
10.
11.  public void setDependentItem(ProjectItem newDependentItem){ dependentItem =
   newDependentItem;  }
12.
13.  public String toString(){
14.     return getProjectItem().toString() + EOL_STRING
15.       + "\tProjectItem dependent on: " + dependentItem;
16. }
17.}
```

Класс SupportedProjectItem (листинг 3.18) "декорирует" класс ProjectItem и сохраняет ссылку на коллекцию ArrayList вспомогательных документов — файловых объектов, представляющих дополнительную информацию или ресурсы.

Листинг 3.18. SupportedProjectItem.java

```
1. import java.util.ArrayList;
2. import java.io.File;
3. public class SupportedProjectItem extends ProjectDecorator{
4.   private ArrayList supportingDocuments = new ArrayList();
5.
6.   public SupportedProjectItem(){ }
7.   public SupportedProjectItem(File newSupportingDocument){
8.      addSupportingDocument(newSupportingDocument);
9.   }
10.
11.  public ArrayList getSupportingDocuments(){
12.     return supportingDocuments(){
13.  }
14.
15.  public void addSupportingDocument(File document){
16.     if (!supportingDocuments.contains(document)){
17.        supportingDocuments.add(document);
18.     }
19.  }
20.
21.   public void removeSupportingDocument(File document){
22.      supportingDocuments.remove(document);
23.  }
24.
25.   public String toString(){
26.     return getProjectItem().toString() + EOL_STRING
27.        + "\tSupporting Documents: " + supportingDocuments;
28.  }
29.}
```

Достоинством именно такого способа определения дополнительных возможностей является то, что он облегчает создание элементов проекта, обладающих некоторой комбинацией возможностей. Используя эти классы, можно создать этап проекта, зависящий от другого элемента проекта, или этап со вспомогательными документами. Можно также объединить все "декорирующие" классы в цепочку и создать этап, который зависит от другого этапа и имеет вспомогательные документы. Эта гибкость и является самой сильной характеристикой шаблона Decorator.

Facade

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Создание упрощенного интерфейса для группы подсистем или сложной подсистемы.

Представление

Каждому пользователю хотелось бы иметь возможность изменения пользовательского интерфейса в соответствии со своими предпочтениями или для того, чтобы сделать свою работу более удобной. Например, некоторые пользователи имеют слабое зрение и испытывают затруднения при чтении слишком мелких шрифтов, в связи с чем им нужно средство, обеспечивающее увеличение размера стандартного шрифта пользовательского интерфейса. Естественно, решение, сводящееся к принуждению пользователя пройти всю процедуру начальной настройки приложения, отвечая на вопросы многочисленных диалоговых окон (причем эти вопросы выводятся тем самым мелким шрифтом, от которого пользователь хочет избавиться!) о предпочитаемых параметрах настройки модема, принтера и сканера, вряд ли можно назвать дружественным. Для этой цели гораздо лучше подойдет специальный мастер, при создании которого были бы учтены все особенности восприятия информации человеком со слабым зрением.

Несмотря на то, что такой мастер оказывает некоторую специфическую помощь, нет никакой необходимости в том, чтобы как-то ограничивать набор параметров, предоставляемых мастером пользователю для настройки. Такой мастер — это лишь особое представление системы, не затрагивающее ее функциональность. Создание подобного мастера происходит в соответствии с шаблоном проектирования, получившего название Facade.

Область применения

Шаблон Facade рекомендуется использовать в следующих случаях.

- Необходимо упростить работу со сложной системой, предоставив в распоряжение пользователя интерфейс, который, с одной стороны, является более простым вариантом стандартного интерфейса, а с другой — сохраняет все параметры тонкой настройки, присутствующие в стандартном интерфейсе.
- Требуется уменьшить зависимость клиентов от подсистем.
- Нужно создать слои подсистем, используя шаблон Facade для наборов подсистем.

Описание

Большинство современных программных систем достаточно сложны. Шаблоны проектирования призваны помочь разработчикам таким образом структурировать приложения, чтобы уменьшить влияние их сложности на процесс разработки и использования. Одним из часто используемых в шаблонах проектирования подходов является распределение функциональности по нескольким небольшим классам. Кроме того, может применяться такой подход, как разделение системы на отдельные части, в результате чего также нередко появляются дополнительные классы. Разделение

системы на несколько подсистем позволяет не только снизить степень сложности, но и разделить процесс разработки на отдельные этапы.

Разделение системы на несколько специализированных классов — это устоявшаяся практика ООП. Однако наличие в системе большого количества классов может быть не только достоинством, но и недостатком.

Клиентам, использующим такую систему, необходимо иметь дело с большим количеством объектов. Любой пользователь, столкнувшись с сотнями конфигурационных параметров, скорее всего придет в замешательство. Это обстоятельство уже давно было понято производителями автомобилей — вспомните, приходилось ли вам хоть раз сталкиваться с необходимостью установки степени обогащенности топливной смеси? Были времена, когда водитель должен был устанавливать этот параметр перед каждым запуском двигателя. Однако сегодня мало кто из водителей догадывается о существовании такого параметра: все, что мы должны сделать, — вставить ключ в замок зажигания и провернуть его, чтобы запустить двигатель. Все остальное выполняет за нас машина. Таким образом, чем меньше параметров контролирует клиент, тем лучше для него. Шаблон Facade может предоставить в распоряжение пользователя все необходимые ему параметры и определить, какие подсистемы нужно вызвать.

Обычно реализация шаблона Facade делегирует большую часть работы подсистемам, но может какую-то часть выполнять и самостоятельно.

Необходимо подчеркнуть, что шаблон Facade предназначен не для *сокрытия* подсистем. Его цель — обеспечить более простой интерфейс для определенного набора подсистем, но так, чтобы не пострадали интересы клиентов, которым требуется полноценный доступ к параметрам тонкой настройки.

Одним из примеров реализации шаблона Facade является часто применяемые на практике мастера настройки.

Реализация

Диаграмма классов шаблона Facade представлена на рис. 3.7.

При реализации шаблона Facade обычно используются следующие классы.

- Facade. Класс, используемый клиентами. Этот класс знает, с какими подсистемами он работает и за что отвечает каждая из этих подсистем. Обычно все запросы клиента делегируются соответствующим подсистемам.
- Subsystem. Набор классов. Клиенты могут работать с такими наборами как напрямую, так и опосредованно, через класс Facade. Подсистемы ничего не знают о существовании последнего, для них класс Facade — это всего лишь еще один клиент.

Достоинства и недостатки

Достоинство шаблона Facade состоит в том, что он предоставляет простой интерфейс для взаимодействия со сложной системой, не уменьшая при этом возможностей управления. Иными словами, данный интерфейс просто защищает клиента от слишком большого количества параметров.

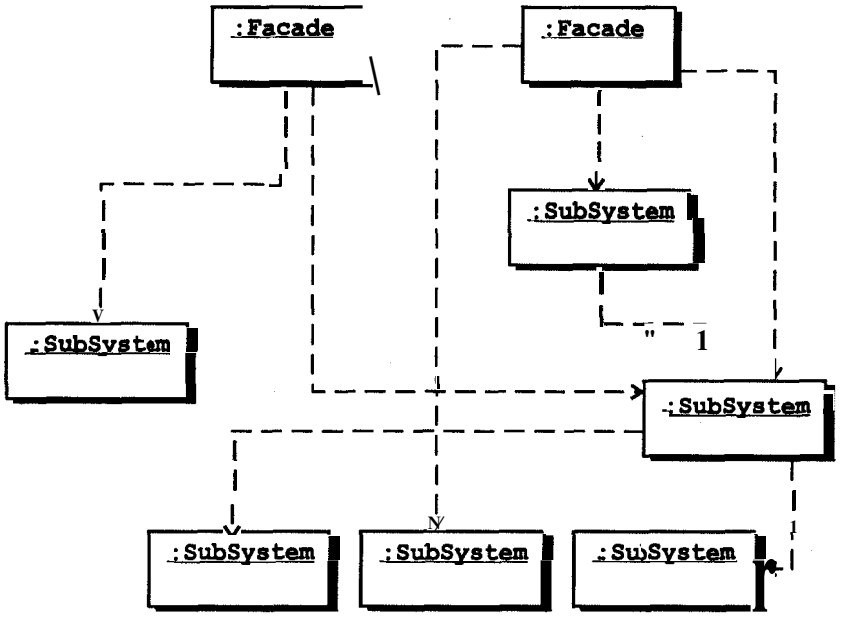


Рис. ***3.7.*** Диаграмма классов шаблона Facade

Шаблон Facade транслирует запросы клиентов тем подсистемам, которые могут их обслужить. В большинстве случаев **один** запрос делегируется сразу нескольким подсистемам. Поскольку клиент взаимодействует лишь с классом Facade, это позволяет менять внутреннюю логику системы, не затрагивая принципов работы клиента с классом Facade.

Применение шаблона Facade способствует снижению взаимосвязи клиента с подсистемами. Кроме того, этот шаблон с успехом можно применять и для снижения взаимосвязей между самими подсистемами. Каждая подсистема может иметь собственный экземпляр класса Facade, через который с ней взаимодействуют классы из других частей системы.

Варианты

Шаблон Facade может реализовываться в следующих вариантах.

- Facade может реализовываться как в виде интерфейса, так и в виде абстрактного класса, что позволяет отложить определение деталей реализации, а также снижает взаимосвязь компонентов системы.
- Для одного и того же набора подсистем можно определить несколько экземпляров класса Facade.
- Шаблон Facade иногда используют для сокрытия подсистем. Когда в архитектуре системы определено, что данный шаблон реализуется на границах систем, это означает, что тем самым достигается снижение сложности взаимодействия систем между собой. Например, система, в которой все вызовы проходят через

централизованный фасадный интерфейс, является более удобной в сопровождении, чем система, содержащая множество взаимодействующих друг с другом напрямую классов.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Abstract Factory (стр. 26). Шаблон Abstract Factory предназначен для создания семейств родственных объектов. Для упрощения доступа к различным объектам, создание которых выполняется с помощью механизма, определенного по шаблону Abstract Factory, этот же механизм может использоваться и для создания фасадного объекта.
- Mediator (стр. 95). Шаблоны Mediator и Facade внешне очень похожи. Различие состоит в их назначении и реализации. Шаблон Mediator позволяет упростить взаимодействие между объектами, что реализуется путем добавления новой функциональности. Шаблон Facade — это просто некая абстракция интерфейса одной или более подсистем.
- Singleton (стр. 54). Шаблон Facade использует шаблон Singleton в тех случаях, когда нужно получить единый, глобально доступный центр, через который осуществляется обращение к подсистеме.
- Session Facade [CJ2EEP]. Это тот же шаблон Facade, в котором инкапсулированы сложные механизмы Enterprise JavaBeans, что упрощает работу с этой системой ее клиентам.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `Run Pattern`, приведен в разделе "Facade" на стр. 462 Приложения А.

Для того чтобы PIM-приложение было с точки зрения пользователей более функциональным, нужно дать им возможность настраивать его параметры. К таким настраиваемым параметрам можно отнести, например, гарнитуру и размер шрифта, цветовую гамму, порядок запуска отдельных подсистем, используемую по умолчанию денежную единицу и т.п. В данном примере рассмотрен набор параметров, определяющих представление в системе национальных стандартов даты, времени, валюты и т.п.

Роль класса Facade в рассматриваемом примере играет класс `InternationalizationWizard` (листинг 3.19). Этот класс координирует взаимодействие клиента с несколькими объектами, так или иначе связанными с выбранным национальным стандартом.

Листинг 3.19. InternationalizationWizard.java

```
1. import java.util.HashMap;
2. import java.text.NumberFormat;
3. import java.util.Locale;
4. public class InternationalizationWizard{
5.   private HashMap map;
6.   private Currency currency = new Currency));
7.   private InternationalizedText propertyFile = new InternationalizedText();
8.
9.   public InternationalizationWizard() {
10.     map = new HashMap();
11.    Nation[] nations = {
12.      new Nation("US", 'S', "+1", "us.properties",
   NumberFopmat.getInstance(Locale.US)),
13.      new Nation("The Netherlands", 'f', "+31', "dutch.properties",
   NumberFormat.getInstance(Locale.GERMANY)),
14.      new Nation("France", 'f', "+33", "french.properties",
   NumberFormat.getInstance(Locale.FRANCE))
15.    );
16.    for (int i = 0; i < nations.length; i++) {
17.      map.put(nations[i].getName(), nations[i]);
18.    }
19. }
20.
21. public void setNation(string name) {
22.    Nation nation = (Nation)map.get(name);
23.    if (nation != null) (
24.       currency.setCurrencySymbol(nation.getSymbol());
25.       currency.setNumberFormat(nation.getSymbol());
26.      PhoneNumber.setSelectedInterPrefix(nation.getDialingPrefix()) ;
27.       propertyFile.setFileName(nation.getPropertyFileName());
28.    }
29. }
30.
31. public Object[] getNations(){
32.    return map.values().toArray();
33. }
34. public Nation getNation(String name)[
35.    return (Nation)map.get(name);
36. }
37. public char getCurrencySymbol(){
38.    return currency.getCurrencySymbol();
39. }
40. public NumberFormat getNumberFormat(){
41.    return currency.getNumberFormat(){
42. }
43. public String getPhonePrefix(){
44.    return ProneNumber.getSelectedInterPrefix();
45. }
46. public String getProperty(string key){
47.    return propertyFile.getProperty(key);
48. }
49. public String getProperty(String key, String defaultValue){
50.    return propertyFile.getProperty(key, defaultValue);
51. }
52.}
```

Обратите внимание на то, что класс InternationalizationWizard имеет несколько методов вида getXXXX, с помощью которых он осуществляет делегирование вызовов ассоциированным объектам. Кроме того, класс имеет метод setNation, используемый для изменения типа национального стандарта в соответствии с предпочтениями пользователя.

Хотя управление параметрами представления национального стандарта, как показано в данном примере, осуществляется через фасадный класс InternationalizationWizard, при этом сохраняется возможность управления каждым объектом в отдельности. Эта особенность является одним из преимуществ данного шаблона — он позволяет в некоторых случаях управлять группой объектов, но вместе с тем не ограничивает свободы по раздельному управлению компонентами.

Вызов метода setNation фасадного класса InternationalizationWizard используется для установки типа выбранного национального стандарта. Выполнение этой операции ведет к изменению параметров классов Currency (листинг 3.20), PhoneNumber (листинг 3.21) и InternationalizedText (листинг 3.22).

Листинг 3.20. Currency.java

```
1. import java.text.NumberFormat;
2. public class Currency{
3.  private char currencySymbol;
4.  private NumberFormat numberFormat;
5.
6.  public void setCurrencySymbol(char newCurrencySymbol){ currencySymbol =
   newCurrencySymbol; }
7.  public void setNumberFormat(NumberFormat newNumberFormat){ numberFormat =
   newNumberFormat; }
8.
9.  public char getCurrencySymbol(){ return currencySymbol; }
10. public NumberFormat getNumberFormat(){ return numberFormat; }
11.}
```

Листинг 3.21. PhoneNumber.java

```
1. public class PhoneNumber {
2.  private static String selectedInterPrefix;
3.  private String internationalPrefix;
4.  private String areaNumber;
5.  private String netNumber;
6.
7.  public PhoneNumber(String intPrefix, String areaNumber, String netNumber) {
8.     this.internationalPrefix = intPrefix;
9.     this.areaNumber = areaNumber;
10.    this.netNumber = netNumber;
11. }
12.
13. public String getInternationalPrefix(){ return internationalPrefix; }
14. public String getAreaNumber(){ return areaNumber; }
15. public String getNetNumber(){ return netNumber; }
16. public static String getSelectedInterPrefix(){ return selectedInterPrefix;
   }
17.
18. public void setInternationalPrefix(String newPrefix){ internationalPrefix =
   newPrefix; }
```

```
19. public void setAreaNumber (String newAreaNumber) { areaNumber =
    newAreaNumber; }
20. public void setNetNumber(String newNetNumber){ netNumber = newNetNumber; }
21. public static void setSelectedInterPrefix(String prefix){
    selectedInterPrefix = prefix; }
22.
23. public String toString(){
24.    return internationalPrefix + areaNumber + netNumber;
25. )
26.}
```

Листинг 3.22. InternationalizedText.java

```
1. import java.util.Properties;
2. import java.io.File;
3. import java.io.IOException;
4. import java.io.FileInputStream;
5. public class InternationalizedText{
6.   private static final String DEFAULT_FILE_NAME = "";
7.   private Properties textProperties = new Properties();
8.
9.   public InternationalizedText(){
10.    this(DEFAULT_FILE_NAME);
11.  }
12.  public InternationalizedText(String fileName){
13.    loadProperties(fileName);
14.  }
15.
16.  public void setFileName(String newFileName){
17.    if (newFileName != null){
18.      loadProperties(fileName);
19.    }
20.  }
21.  public String getProperty(String key){
22.   return getProperty(key, " " ) ;
23.  }
24.  public String getProperty(String key, String defaultValue){
25.    return textProperties.getProperty(key, defaultValue);
26.  }
27.
28.  private void loadProperties(String fileName){
29.    try{
30.      FileInputStream input = new FileInputStream(fileName);
31.      textProperties.load(input);
32.    }
33.    catch (IOException exc){
34.     textProperties = new Properties ();
35.    }
36.  }
37.}
```

Общие данные о стране хранятся вспомогательным классом Nation (листинг 3.23). При создании первого экземпляра класса InternationalizationWizard последний создает коллекцию объектов Nation.

Листинг 3.23. Nation.java

```
1. import java.text.NumberFormat;
2. public class Nation {
3.   private char symbol;
4.   private String name;
5.   private String dialingPrefix;
6.   private String propertyFileName;
7.   private NumberFormat numberFormat;
8.
9.   public Nation (String newName, char newSymbol, String newDialingPrefix,
10.     String newPropertyFileName, NumberFormat newNumberFormat)  (
11.     name = newName;
12.     symbol = newSymbol;
13.     dialingPrefix = newDialingPrefix;
14.     propertyFileName = newPropertyFileName;
15.     numberFormat = newNumberFormat;
16. }
17.
18.  public String getName() { return name; }
19.  public char getSymbol0 ( return symbol; }
20.  public String getDialingPrefix(){ return dialingPrefix; }
21.  public String getPropertyFileName(){ return propertyFileName; }
22.  public NumberFormat getNumberFormat(){ return numberFormat; }
23.
24.  public String toString() { return name; }
25.}
```

Flyweight

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Уменьшение количества объектов системы с многочисленными низкоуровневыми особенностями путем совместного использования подобных объектов.

Представление

Применение на практике принципов объектно-ориентированного программирования ведет к тому, что во время выполнения приложения создается множество объектов (особенно ярко эта особенность проявляется в тех случаях, когда в приложении интенсивно используются низкоуровневые объекты). Это приводит к увеличению накладных расходов на использование памяти виртуальной машины Java (JVM — Java Virtual Machine).

Поскольку многие объекты PIM-приложения поддерживают редактирование, они выполняются с использованием шаблона State (подробнее см. раздел "State" на стр. 120), что позволяет таким объектам определять, нужно ли сохранять содержимое

редактируемого элемента. При этом каждый элемент может иметь собственную коллекцию объектов-состояний.

Одним из способов избавления от необходимости задействовать множество объектов является совместное их использование. Действительно, многие из этих низкоуровневых объектов отличаются один от другого лишь в незначительных деталях, тогда как большая часть данных и логики таких объектов остается неизменной. Совместное использование экземпляров соответствующего класса позволяет резко снизить количество объектов в системе, ничуть не теряя ее функциональности.

Применение шаблона Flyweight позволяет для определенного набора объектов отделить общую часть всех объектов от частностей. Данные, которыми отличаются один от другого различные экземпляры класса (их иногда называют экстернальными), подключаются к типовому экземпляру по мере необходимости.

Область применения

Шаблон Flyweight можно использовать в тех случаях, когда выполняются все следующие условия.

- В приложении имеется множество одинаковых (или практически одинаковых) объектов.
- Если объекты практически одинаковы, их различающиеся части можно отделить от одинаковых частей, чем обеспечивается возможность совместного использования последних.
- После отделения различающихся частей, группы почти одинаковых объектов можно заменить одним совместно используемым объектом.
- Приложению необходимо различать практически одинаковые объекты в их исходном состоянии.

Описание

Шаблон Flyweight предназначен для уменьшения количества объектов в приложении путем совместного их использования. Конечно, все объекты содержат некоторые внутренние данные, однако все данные, относящиеся к контексту, в котором они работают, предоставляются таким объектам извне. Каждый совместно используемый объект должен обладать как можно меньшей индивидуальностью и быть независимым от внешних источников с точки зрения своего содержимого.

Совместное использование объектов в соответствии с шаблоном Flyweight, уменьшает количество объектов. Совместно используемый объект взаимодействует с несколькими клиентами и, с точки зрения последних, ничем не отличается от обычного объекта.

Примером реализации шаблона Flyweight является диспетчер компоновки (layout manager). При разработке GUI, как правило, задействуется несколько компонентов и контейнеров. Для определения визуальной компоновки графического интерфейса применяются диспетчеры компоновки. По большому счету все диспетчеры подобны один другому, их различия проявляются лишь в том, какими компонентами они управ-

ляют, а также в некоторых атрибутах. Если убрать такие компоненты и атрибуты, любой экземпляр любого диспетчера компоновки не будет отличаться от других экземпляров. Когда возникает необходимость в функциональности конкретного диспетчера компоновки, компоненты и атрибуты передаются одному совместно используемому экземпляру. Совместное использование лишь одного объекта для всех типов диспетчеров компоновки с настройкой его на нужный контекст, позволяет уменьшить количество объектов системы.

Клиенты, работающие с совместно используемыми объектами, отвечают за предоставление и (или) вычисление информации, относящейся к их контексту. Дополнительная информация поступает совместно используемому объекту по мере необходимости.

Поскольку типовой объект используется совместно, клиенту не нужно создавать его самому. Вместо этого он получает такой объект через специальный механизм генерации объектов (подробнее см. раздел "Abstract Factory" на стр. 26). Именно этот механизм и гарантирует корректное совместное использование объектов, создаваемых в соответствии с шаблоном Flyweight.

Однако далеко не все объекты, полученные в результате применения шаблона Flyweight, должны обязательно быть совместно используемыми. То же самое относится и к реализующим классам. Шаблон Flyweight лишь позволяет организовать совместное использование, но он не навязывает его.

Использовать шаблон Flyweight лучше всего в тех случаях, когда можно легко идентифицировать и отделить внешние данные от объектов (при этом количество состояний должно быть достаточно ограниченным).

Реализация

Диаграмма классов шаблона Flyweight представлена на рис. 3.8.

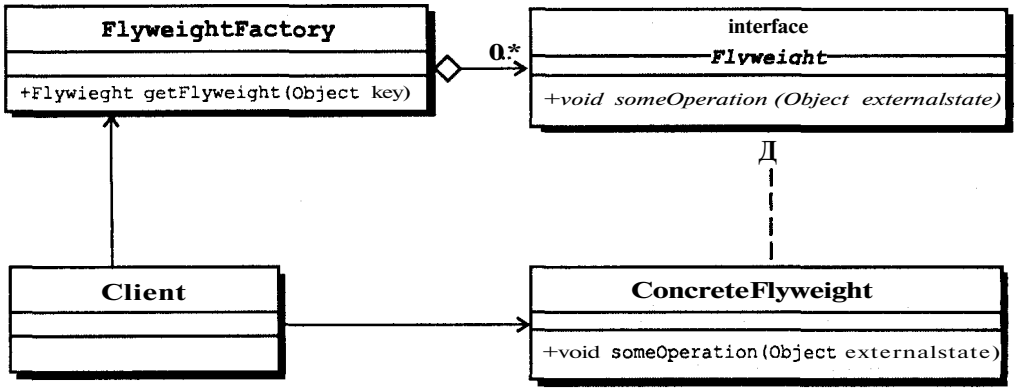


Рис. *3.8.* Диаграмма классов шаблона Flyweight

При реализации шаблона Flyweight обычно используются следующие классы.

- Flyweight. Интерфейс, определяющий методы, с помощью которых клиенты могут передать внешнее состояние типовым объектам.

- `ConcreteFlyweight`. Реализация интерфейса Flyweight, которая обеспечивает возможность хранения внутренних данных. Внутренние данные должны быть репрезентативными для всех совместно используемых объектов.
- FlyweightFactory (см. раздел "Abstract Factory" на стр. 26). Механизм, отвечающий за создание типовых объектов и управление ими. Обеспечение доступа к экземпляру класса Flyweight посредством специального механизма его создания гарантирует корректность совместного доступа. Такой механизм может генерировать сразу все типовые объекты в момент запуска приложения или выполнять такие операции по мере необходимости.
- Client. Клиент отвечает за создание контекста для совместно используемого объекта и за передачу ему этого контекста. Единственным способом получения ссылки на совместно используемый объект для клиента является вызов FlyweightFactory.

Достоинства и недостатки

Очевидным достоинством этого шаблона является уменьшение количества обрабатываемых объектов. Это может помочь сэкономить немало памяти, как оперативной, так и дисковой, если сведения об объектах хранятся на диске.

Однако самая ощутимая экономия обеспечивается в тех случаях, когда контекстная информация, необходимая для работы совместно используемым объектам, не сохраняется, а вычисляется. Однако при этом есть и обратная сторона: за такую экономию приходится расплачиваться производительностью.

Теперь вместо хранения множества объектов, клиентам достаточно вычислить контекст и передать его экземпляру совместно используемого объекта. Получив такую информацию, последний производит определенные вычисления или выполняет другие операции. При корректной реализации снижение количества объектов неизбежно проявляется в повышении производительности системы. Необходимо подчеркнуть, что особенно ощутимый выигрыш заметен в тех случаях, когда контекстная информация достаточно невелика, а совместно используемые объекты — наоборот, достаточно объемны.

Варианты

Отсутствуют.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Abstract Factory (стр. 26). Шаблон Abstract Factory применяется для обеспечения доступа к типовым объектам, тем самым обеспечивая корректное совместное использование последних.
- Composite (стр. 171). Этот шаблон часто используется для получения определенной структуры объектов.

- State (стр. 120). Реализация шаблона State нередко осуществляется с использованием шаблона Flyweight.
- Strategy (стр. 130). Это еще один шаблон, при реализации которого удобно воспользоваться преимуществами, предоставляемыми шаблоном Flyweight.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом RunPattern, приведен в разделе "Flyweight" на стр. 470 Приложения ***А.***

В данном примере шаблон Flyweight применяется для организации совместного использования объектов-состояний PIM-приложения. В примере, который был посвящен шаблону State, подобные объекты использовались для редактирования и хранения информации для набора объектов класса Appointment. В настоящем примере объекты-состояния задействованы для управления командами редактирования и сохранения нескольких коллекций объектов.

Интерфейс State (листинг 3.24) определяет стандартное поведение всех объектов-состояний приложения. В этом интерфейсе определены два базовых метода edit и save.

Листинг 3.24. State.java

```
1. package flyweight.example;
2.
3. import java.io.File
4. import java.io.IOException;
5. import java.io.Serializable;
6.
7. public interface State {
8.   public void save(File f, Serializable s) throws IOException;
9.   public void edit();
10.}
```

Интерфейс реализуется двумя классами: CleanState (листинг 3.25) и DirtyState (листинг 3.26). В рассматриваемом примере данные классы применяются для отслеживания состояния нескольких объектов, в связи с чем необходим вспомогательный класс, который бы управлял тем, какой элемент нуждается в обновлении.

Листинг 3.25. CleanState.java

```
1. import java.io.File;
2. import java.io.FileOutputStream;
3. import java.io.IOException;
4. import java.io.ObjectOutputStream;
5. import java.io.Serializable;
6.
```

```
7. public class CleanState implements State{
8.  public void save(File file, Serializable s, int type) throws IOException{ }
9.
10. public void edit(int type){
11.   StateFactory.setCurrentState(StateFactory.DIRTY) ;
12.    ((DirtyState)StateFactory.DIRTY) .incrementStateValue(type);
13. }
14.}
```

Листинг 3.26. DirtyState.java

```
1. package flyweight.example;
2.
3. import java.io.File;
4. import.java.io.FileOutputStream;
5. import.java.io.IOException;
6. import.java.io.ObjectOutputStream;
7. import.java.io.Serializable;
8.
9. public class DirtyState implements State {
10.  public void save(File file, Serializable s) throws IOException {
11.     //сериализация
12.     FileOutputStream fos = new FileOutputStream(file);
13.     ObjectOutputStream out = new ObjectOutputStream(fos);
14.    out.writeObject(s) ;
15. }
16.
17.  public void edit O  {
18.     //опущено
19. }
20.}
```

Поскольку эти два класса применяются для отслеживания общего состояния приложения, ими управляет класс StateFactory (листинг 3.27), который по мере необходимости создает объекты обоих классов.

Листинг 3.27. StateFactory.java

```
1. public class StateFactory (
2.   public static final State CLEAN = new CleanState();
3.   public static final State DIRTY = new DirtyState ();
4.   private static State currentState = CLEAN;
5.
6.   public static State getCurrentState (){
7.      return currentState;
8.   }
9.
10.  public static void setCurrentState(State state){
11.     currentState = state;
12.  }
13.}
```

Half-Object Plus Protocol (HOPP)

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Предоставление единой сущности, которая размещается в двух или более областях адресного пространства.

Представление

Распределенное Java-приложение размещает свои объекты в различных адресных пространствах, предоставляемых виртуальными машинами Java. Некоторые технологии, такие как RMI, даже позволяют удаленным объектам обращаться к методам, которые на самом деле находятся в другой JVM, тем самым обеспечивая не только распределение объектов, но и распределение их поведения и информации об их состоянии. Независимо от того, какая технология используется, объектам, находящимся в разных JVM, нужно как-то взаимодействовать, чтобы все приложение работало как единое целое. Если же такого взаимодействия нет, приложение "разваливается" на отдельные составляющие.

Предположим, на машинах А и Б работают экземпляры JVM. Объекту, выполняющемуся в адресном пространстве JVM А, требуется для обеспечения доступа к методам объекта JVM Б *слепок* (stub) объекта Б (см. раздел "Proxy" на стр. 209). Получить слепок объекта можно несколькими способами. Не вдаваясь в подробности, отметим, что после того, как объект А получит требуемый слепок, он может вызывать методы этого слепка, и все подобные вызовы будут транслироваться объекту Б.

Недостатком данного подхода является то, что по сети требуется пересылать вызовы *всех без исключения* методов, что далеко не всегда можно назвать удачным решением. Иногда требуется, чтобы какие-то методы слепка выполнялись локально, без передачи их вызовов удаленному объекту.

Можно сделать вывод о том, что в таких случаях требуется наличие объекта, который находится одновременно в двух или более адресных пространствах. Подобный *прокси-объект* (proxy), являющийся некоторым представлением удаленного объекта, рассматривается как часть последнего. Такой объект может вызывать и локальные, и удаленные методы, реализуя свою функциональность в нескольких адресных пространствах. Работа подобного объекта строится в соответствий с шаблоном HOPP.

Область применения

Шаблон HOPP рекомендуется использовать в следующих случаях.

- Объект должен находиться одновременно в двух различных адресных пространствах в виде единой сущности.

- Какие-то методы должны выполняться удаленно, а какие-то — только локально.
- Необходимо обеспечить прозрачное для вызывающего объекта применение способов оптимизации, таких как кэширование или объединение нескольких запросов в единую сетевую транзакцию.

Описание

Разработка распределенных приложений — задача не из легких. Одна из проблем, с которыми приходится сталкиваться при создании таких приложений, состоит в том, что требуется размещать единые сущности (объекты) в нескольких адресных пространствах. Такое размещение объектов диктуется, например, необходимостью получения доступа к нескольким физическим устройствам, установленным на разных машинах, или объясняется невозможностью логического разделения объекта.

Для разделения объекта на две части, которые будут взаимодействовать друг с другом через удаленное соединение, можно воспользоваться командой rmic с параметром -keep или -keepgenerated. Это позволяет получить исходный код слепка. Затем требуется отредактировать исходный код таким образом, чтобы определенные методы не пересылались удаленному объекту, а обрабатывались самим слепком. Редактирование завершается компиляцией, по выполнении которой в распоряжении разработчика оказывается собственная модифицированная версия слепка.

К сожалению, данный путь решения проблемы нельзя назвать идеальным, так как он лишает разработчика возможности пользоваться командой rmic. Действительно, каждый раз при выполнении этой команды по отношению к удаленному объекту придется снова и снова заниматься редактированием получаемого кода.

Лучше всего каким-то образом на уровне представления разделить объект на две части и обеспечить взаимодействие между этими частями. Каждую половину при этом необходимо реализовать таким образом, чтобы она могла взаимодействовать с объектами, находящимися в ее адресном пространстве. Ответственность за синхронизацию обоих половин и пересылку информации в обоих направлениях можно возложить на специальный протокол.

Именно так и работает шаблон HOPP — в соответствии с этим шаблоном создается объект, реализующий требуемый удаленный интерфейс и содержащий ссылку на оригинальный слепок удаленного объекта. Вызовы методов, которые должны обрабатываться обычным образом (т.е. отправляться удаленному объекту), пересылаются слепку, а вызовы методов, которые должны выполняться локально, обрабатываются новым классом.

Название HOPP получается из сокращения Half-Object Plus Protocol : с точки зрения клиента, объект разделен на две части и ему достается лишь одна половина этого объекта (half of the object). Кроме того, в его распоряжении имеется специальный протокол (protocol), привязанный к этой половине и с помощью которого осуществляется взаимодействие со второй половиной объекта. Отсюда и полное название шаблона: Half-Object Plus Protocol .

Реализация

Диаграмма классов шаблона HOPP представлена на рис. 3.9.

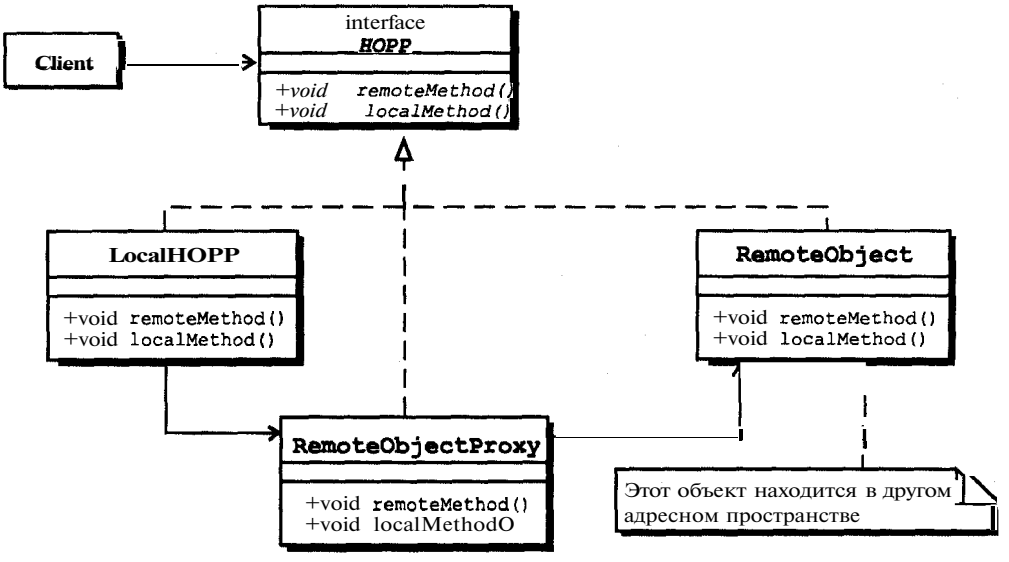


***Рис.** 3.9.* Диаграмма классов шаблона HOPP

При реализации шаблона HOPP обычно используются следующие классы.

- HOPP. Данный интерфейс определяет методы, которые должны быть доступными для клиентов. Обе половины объекта, выполненного в соответствии с шаблоном HOPP, реализуют этот интерфейс.
- LocalHOPP. Класс, реализующий интерфейс HOPP. Некоторые методы класса выполняются локально, а вызовы остальных передаются классу `RemoteObject-Proxy`.
- `RemoteObj ectProxy`. Этот класс играет роль удаленного прокси-объекта, пересылающего все вызовы другой половине объекта, которая находится в ином адресном пространстве. Данный прокси-объект инкапсулирует в себе протокол, связывающий две половины объектов.
- `RemoteOb ject`. Это половина выполненного в соответствии с шаблоном HOPP объекта, содержащая все удаленно выполняемые методы.
- `Client`. Этот класс вызывает методы интерфейса HOPP. Эти вызовы методов прозрачны для клиента независимо от того, обращается ли он к удаленному прокси-объекту (см. раздел "Proxy" на стр. 209), объекту HOPP или локальному объекту.

Достоинства и недостатки

Достоинством этого шаблона является то, что он позволяет без особых накладных расходов получить единый объект, находящийся одновременно в двух адресных пространствах. Для клиентов, использующих одну часть такого объекта, все его особенности остаются прозрачными. Иными словами, клиентам совершенно не нужно знать, находится ли объект, с которым они работают, в одном адресном пространстве или в нескольких.

Более того, различия можно скрыть до такой степени, что у клиентов может сложиться впечатление, что они используют "чистый" локальный объект, тогда как на самом деле часть этого объекта будет удаленной. Эту часть можно реализовать таким образом, чтобы клиенты считали, что они работают с удаленным прокси-объектом, а в действительности они будут работать с разделенным объектом, содержащим удаленный прокси-объект. Это позволяет получить дополнительный выигрыш в том, что некоторые методы, предназначенные для удаленного выполнения, будут выполняться локально.

Очень важным преимуществом данного шаблона является возможность повышения производительности. Каждая половина объекта может определять, когда и как он будет обмениваться информацией с другой половиной. Наличие таких коммуникационных стратегий позволяет улучшить производительность путем снижения количества сетевых вызовов методов, причем без вмешательства клиента как на одной, так и на другой стороне.

Недостаток шаблона состоит в дублировании определенной части функциональности. Это объясняется тем, что каждая половина объекта должна иметь возможность обрабатывать локальные объекты.

Варианты

Шаблон HOPP может реализовываться в следующих вариантах.

- Каждая половина объекта хранит ссылку на другую половину и обменивается с ней сообщениями, пересылаемыми в обоих направлениях. В классической форме реализации разделенного объекта такая ссылка содержится только в той половине, которая находится на стороне клиента. Это приводит к тому, что коммуникации могут осуществляться лишь по инициативе клиентской части объекта путем вызова метода, принадлежащего удаленной половине. Удаленная половина может ответить на каждый запрос только один раз, возвращая при этом определенное значение. Но возможны ситуации, в которых обе половины разделенного объекта могут инициировать обмен данными, в связи с чем обоим нужно иметь ссылку на другую половину.
- *SmartHOPP (SHOPP).* В этой реализации локальная часть разделенного объекта может выбрать вторую половину с помощью одной из нескольких стратегий подключения. Данное решение может оказаться полезным в тех случаях, когда, например, приложение распределено в реконфигурируемой сети, узлы которой постоянно отключаются и подключаются.
- *AsymmetricHOPP.* В данном варианте шаблона HOPP снимается требование обязательности реализации обеими половинами распределенного объекта одного

и того же интерфейса. Удаленная часть может представлять собой новый прокси-объект, услугами которого пользуется локальная часть разделенного объекта. Новый прокси-объект может обеспечивать новые способы оптимизации или даже быть прокси-объектом для совершенно другого удаленного объекта.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Mediator (стр. 95). Представляет объекты, распределенные по нескольким адресным пространствам. Для упрощения взаимодействия таких объектов шаблон Mediator может использовать шаблон HOPP.
- Proxy (особенно в варианте удаленного прокси-объекта, см. стр. 209). Шаблон HOPP пользуется шаблоном Proxy для обеспечения прозрачного обмена информацией между двумя половинами разделенного объекта.

Пример

Примечание

Полный работающий код данного примера *со* ***вспомогательными классами,*** *а* ***также классом*** *RunPattern, приведен в разделе* ***"Half-Object Plus Protocol (HOPP)"*** *на стр. 475 Приложения* ***А.***

PIM-приложение должно быть доступно с любого рабочего места, но его данные должны храниться централизованно. В данном примере используется шаблон HOPP и технология RMI, обеспечивающие хранение личного календаря на сервере с одновременной поддержкой возможности обращения удаленных пользователей к информации календаря.

Интерфейс Calendar (**листинг 3.28**) определяет все методы, которые должны быть доступны в качестве удаленных. Этот интерфейс расширяет интерфейс java.rmi.Remote, а все его методы генерируют особые ситуации класса java.rmi.RemoteException. В данном примере интерфейс Calendar определяет три метода: getHost, getAppointments и addAppointment.

Листинг 3.28. Calendar.java

```
1. import java.rmi.Remote;
2. import.java.rmi.RemoteException;
3. import.java.util.Date;
4. import java.util.ArrayList;
5. public interface Calendar extends Remote{
6.   public String getHost() throws RemoteException;
7.   public ArrayList getAppointments(Date date) throws RemoteException;
8.   public void addAppointment(Appointment appointment, Date date) throws
     RemoteException;
9. }
```

Интерфейс Calendar реализуется двумя классами: удаленным RMI-объектом и его слепком, или прокси-объектом (см. раздел "Proxy" на стр. 209). Класс удаленного объекта `CalendarImpl` (листинг 3.29) содержит реализацию методов, а слепок управляет взаимодействием с удаленным объектом. Для генерации слепка и опорного класса (skeleton class) нужно воспользоваться командой запуска компилятора RMI Java (`rmic`), указав ей в качестве параметра имя класса CalendarImpl. Опорный класс генерируется для обеспечения обратной совместимости, но начиная с Java 1.2, необходимость в нем уже не возникает.

Листинг 3.29. `CalendarImpl.java`

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. import java.io.File;
4. import java.util.Date;
5. import java.util.ArrayList;
6. import java.util.HashMap;
7. public class CalendarImpl implements Calendar{
8.   private static final String REMOTE SERVICE = "calendarimpl";
9.   private static final String DEFAULT FILE NAME = "calendar.ser";
10.  private HashMap appointmentCalendar = new HashMap();
11.
12.  public CalendarImpl(){
13.     this(DEFAULT FILE NAME);
14. }
15.  public CalendarImpl(String filename)(
16.     File inputFile = new File(filename);
17.     appointmentCalendar = (HashMap)FileLoader.loadData(inputFile);
18.     if (appointmentCalendar == null){
19.       appointmentCalendar = new HashMapO;
20.     }
21.    try f
22.       UnicastRemoteObject.exportObject(this);
23.       Naming.rebind(REMOTE SERVICE, this);
24.     )
25.     catch (Exception exc){
26.       System.err.println("Error using RMT to register the CalendarImpl " +
   exc);
27.     )
28. }
29.
30.  public String getHost(){ return ""; J
31.  public ArrayList getAppointments(Date date){
32.     ArrayList returnValue = null;
33.     Long appointmentKey = new Long(date.getTime();
34.     if (appointmentCalendar.containsKey(appointmentKey)){
35.       returnValue = (ArrayList)appointmentCalendar.get(appointmentKey);
36.     >
37.     return returnValue;
38. }
39.
40.  public void addAppointment(Appointment appointment, Date date){
41.     Long appointmentKey = new Long(date.getTime());
42.     if (appointmentCalendar.containsKey(appointmentKey)){
43.       ArrayList appointments =
   (ArrayList)appointmentCalendar.get(appointmentKey);
44.       appointments.add(appointment);
45.     }
46.     else {
```

```
47.      ArrayList appointments = new ArrayList();
48.       appointments.add(appointment);
49.      appointmentCalendar.put(appointmentKey, appointments);
50.     }
51. }
52.}
```

Для того чтобы иметь возможность обрабатывать входящие коммуникационные запросы, объект класса `CalendarImpl` должен использовать вспомогательный класс RMI `UnicastRemoteObject`. В рассматриваемом примере конструктор класса Calendarlmpl экспортирует сам себя, используя статический метод `UnicastRemoteObject.exportObject`.

Кроме того, класс Calendarlmpl каким-то образом должен обеспечить доступ к себе извне. Для этого служба имен RMI вызывает `rmiregistry`. Эта команда должна быть выполнена до того, как будет создан объект класса Calendarlmpl. Работа команды `rmiregistry` строится по принципу телефонного справочника, поскольку она обеспечивает связь между объектом и его именем. Когда объект Calendarlmpl регистрирует себя с помощью команды `rmiregistry`, вызывая метод `rebind`, последний связывает имя "`calendarimpl`" со слепком соответствующего удаленного объекта.

Для того чтобы клиент смог использовать удаленный объект, ему нужно выполнить поиск в реестре RMI соответствующего узла сети и получить слепок нужного ему объекта. Слепок в данном случае можно сравнить с номером телефона — этот номер можно использовать из любой точки, воспользовавшись любым телефонным аппаратом, а после установления связи вы будете общаться с любым человеком, который возьмет трубку. В настоящем примере в качестве клиента объекта Calendarlmpl выступает класс `CalendarHOPP` (листинг 3.30).

Листинг 3.30. CalendarHOPP.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. import java.util.Date;
4. import java.util.ArrayList;
5. public class CalendarHOPP implements Calendar, java.io.Serializable{
6.  private static final String PROTOCOL = "rmi://";
7.  private static final String REMOTE SERVICE - "/calendarimpl";
8.  private static final String HOPP SERVICE = "calendar";
9.  private static final String DEFAULT HOST = localhost";
10.  private Calendar calendar;
11.  private String host;
12.
13.  public CalendarHOPP(){
14.     this(DEFAULT HOST);
15.  )
16.  public CalendarHOPP(String host){
17.     try {
18.       this.host - host;
19.       String url = PROTOCOL + host + REMONE_SERVICE;
20.       calendar =  (Calendar)Naming.lookup(url);
21.       Naming.rebind(HOPP_SERVICE,  this):
22.     }
23.    catch (Exception exc) {
24.      System.err.println("Error using RMI to look up the Calendarlmpl or
   register the CalendarHOPP " + exc);
```

```
25.     }
26.  }
27.  public String getHost()( return host: ]
28.  public ArrayList getAppointments(Date date) throws RemoteException{ return
   calendar.getAppointments(date); }
29.
30.
31.  public void addAppointment(Appointment appointment, Date date) throws
   RemoteException { calendar.addAppointment(appointment, date); }
32.}
```

Класс CalendarHOPP имеет одно существенное отличие от обычного клиента RMI — он может запускать локально методы, которые в общем случае предназначены для удаленного запуска. Это может оказаться весьма важным преимуществом с точки зрения снижения накладных расходов при обмене данными. Этот класс реализует тот же удаленный интерфейс Calendar, но не экспортирует себя. Он содержит ссылку на слепок и перенаправляет последнему все вызовы методов, которые данный класс не может обработать или не должен обрабатывать. Обеспечив такую пересылку, разработчик может реализовать в рассматриваемом классе те методы, которые, по мнению разработчика, должны выполняться локально. В данном примере к таким методам относится метод getHost. Объект данного класса можно зарегистрировать в реестре RMI подобно тому, как регистрируется обычный слепок. Единственное отличие заключается в том, что данный класс может выполнять методы локально.

Proxy

Также известен как Surrogate

Свойства шаблона

Тип: структурный
Уровень: компонент

Назначение

Представление другого объекта, обусловленное необходимостью доступа или повышения скорости, либо соображениями безопасности.

Представление

По мере продвижения пользователя по службе его PIM-приложению приходится справляться с хранением информации о все большем количестве запланированных встреч и важных дат. Адресная книга пользователя полна адресов знакомых и коллег, включая информацию об их семьях, увлечениях и другие данные, которые в той или иной ситуации могут оказаться важными. Количество хранимых контактов, которое поначалу исчислялось десятками, давно уже перевалило за тысячу.

Однако при этом пользователю нередко приходится открывать PIM-приложение лишь для того, чтобы слегка скорректировать время проведения собрания, сделать заметку о необходимости покупки какой-нибудь мелочи или выполнить какую-нибудь другую сравнительно простую задачу. Понятно, что загрузка всей адресной книги при каждом запуске приложения в таких ситуациях — ненужная и даже раздражающая операция. Пользователю приходится ждать, пока проинициализируются все записи открываемой всякий раз книги, причем никакой пользы из функциональности, порождающей эту задержку, он не извлекает.

С точки зрения пользователя, такую ситуацию нельзя назвать нормальной: все, что ему нужно, — лишь адресная книга, появляющаяся там, где она необходима, и тогда, когда в ней возникает необходимость. (В идеальном случае адресная книга должна появляться за миг *до* того, как пользователь о ней подумает, не раньше и не позже.) Кроме того, необходимость использования в какой-то момент адресной книги вовсе не подразумевает, что пользователю нужна исчерпывающая информация. Например, пользователю могут понадобиться лишь итоговые сведения о количестве хранящихся в книге записей или ему может потребоваться просто добавить в адресную книгу новую запись, не просматривая и не редактируя всей книги. Иными словами, в большинстве случаев пользователю нужна лишь какая-то малая часть информации адресной книги.

Решением данной проблемы является введение промежуточного объекта, который обеспечивает интерфейс с адресной книгой или ее частью. С точки зрения пользователя, этот промежуточный объект выглядит точно так же, как и адресная книга, но его загрузка и отображение выполняется значительно быстрее. С другой стороны, когда пользователю нужна сама адресная книга для выполнения такой операции, как, например, обновление адреса коллеги, промежуточный объект создает для пользователя объект полноценной адресной книги, обеспечивающий выполнение поставленной задачи. Такой промежуточный объект часто называют прокси-объектом в соответствии с названием шаблона Proxy.

Область применения

Шаблон Proxy рекомендуется использовать в тех случаях, когда нужно разработать способ взаимодействия с объектом, работающий по более сложному сценарию, чем при использовании обычной ссылки.

- *Удаленный прокси-объект (remote proxy).* Необходим локальный представитель объекта, который находится в другом адресном пространстве (т.е. выполняется в другой JVM).
- *Виртуальный прокси-объект (virtual proxy).* Требуется объект, который играет роль промежуточного объекта, позволяющего отложить выполнение операции по созданию полноценного объекта (именно этот вариант был описан в предыдущим разделе).
- *Защитный прокси-объект (protection proxy).* Нужно создать специальный объект, который определял бы права доступа к реальному объекту.

Описание

Прокси-объект, или *слепок* (stub), — это представитель другого объекта. Для того чтобы прокси-объект мог представлять некий реальный объект, прокси-объект должен реализовывать точно такой же интерфейс, что и последний. Более того, прокси-объекту требуется хранить ссылку на реальный объект. Необходимость сохранения такой ссылки объясняется тем, что прокси-объект должен уметь в случае необходимости вызывать методы реального объекта. Клиенты взаимодействуют с прокси-объектом, но сам прокси-объект может делегировать выполнение операций реальному объекту. Несмотря на то, что прокси-объект реализует тот же интерфейс, что и реальный объект, он, в отличие от последнего, может решать дополнительные задачи, такие как осуществление удаленных коммуникаций или обеспечение безопасности.

Предположим, прокси-объект — это нечто вроде дублера реального объекта. Реализацию шаблона Proxy можно сравнить с процессом киносъемки опасных трюков. Прокси-объект — это каскадер, а реальный объект — кинозвезда. Во время съемки опасного эпизода из самолета без парашюта прыгает не реальный, а прокси-объект. Поскольку последний реализует точно такой же интерфейс, что и реальный объект, зрители не замечают никаких различий и полагают, что из самолета выпрыгнул реальный объект. Но когда камера берет крупный план (т.е. по сюжету фильма реальный объект должен проявить актерское мастерство), прокси-объект вызывает для выполнения этой работы реальный объект.

Прокси-объекты могут быть самыми разными в зависимости от того, каким правилам они подчиняются.

- *Remote proxy*. Удаленный прокси-объект отвечает за обмен информацией по сети. Он занимается маршелизацией (упаковкой) и демаршелизацией (распаковкой) всех отправляемых и получаемых параметров.
- *Virtual proxy*. Создание реального объекта сопряжено со значительными накладными расходами, поэтому лучше оттягивать эту операцию на как можно более поздний момент или же выполнять ее не сразу, а поэтапно. Чем больше информации известно виртуальному прокси-объекту, тем ниже вероятность того, что понадобится создавать экземпляр реального объекта с целью обеспечения доступа к тем или иным переменным.
- *Protection proxy*. Защитный прокси-объект может использоваться для управления методами доступа, а также для предоставления индивидуальных разрешений для вызова того или иного метода.

Реализация

Диаграмма классов шаблона Proxy представлена на рис. 3.10.

При реализации шаблона Proxy обычно используются следующие классы.

- `Service`. Интерфейс, который реализуют и реальный, и прокси-объект.
- `ServiceProxy`. Класс, который реализует интерфейс `Service` и в случае необходимости перенаправляет все вызовы методов реальному объекту класса `ServiceImpl`.
- `ServiceImpl`. Реальная полноценная реализация интерфейса. Этот объект представляется прокси-объектом класса `ServiceProxy`.

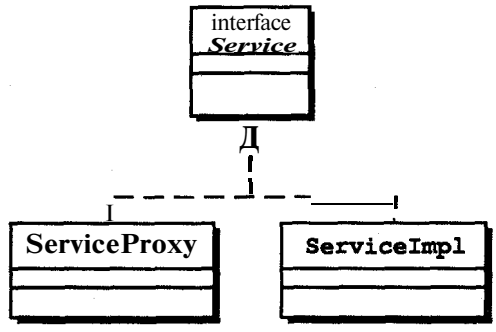


***Рис.** 3.10.* Диаграмма классов шаблона Proxy

Достоинства и недостатки

Последствия, которые проявятся в приложении после реализации шаблона Proxy, в значительной степени определяются типом прокси-объекта.

- *Remote proxy.* Достоинство удаленного прокси-объекта состоит в том, что он позволяет скрыть факт работы в сети от клиента. Клиент, работая с таким объектом, будет считать, что объект локальный. В действительности он имеет дело с локальным объектом, который для выполнения указанных ему операций отправляет вызовы по сети. Необходимо напомнить, что потенциальным недостатком прокси-объектрв такого типа является возможное резкое увеличение времени отклика, что для клиента, не подозревающего о том, что он работает в сети, может оказаться довольно неожиданным.
- *Virtual proxy.* Значительным преимуществом прокси-объектов данного типа является то, что они позволяют работать с собой как с реальными объектами, откладывая создание последних до того момента, когда это действительно понадобится. Более того, такие прокси-объекты могут даже в какой-то мере заниматься оптимизацией, например, определять, когда и как должен создаваться реальный объект.
- *Protection proxy.* Достоинство таких прокси-объектов состоит в том, что они позволяют определять методы управления доступом.

Варианты

Один из вариантов реализации шаблона Proxy состоит в создании такого прокси-объекта, который ничего не знает о реальном объекте, кроме его интерфейса. Это позволяет повысить гибкость системы, но на практике такой вариант можно применять только в тех случаях, когда прокси-объект не отвечает за создание и (или) уничтожение реального объекта.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Adapter (стр. 156). Подобно шаблону Proxy, этот шаблон обеспечивает интерфейс для определенного объекта. Различие состоит в том, что шаблон Proxy предоставляет точно такой же интерфейс, тогда как шаблон Adapter — несколько иной.
- HOPP (стр. 202). Этот шаблон может применять шаблон Proxy для организации взаимодействия между двумя половинами разделенного объекта.
- Business Delegate [CJ2EEP]. Данный шаблон может использоваться для тех же целей, что и шаблон Proxy: объект, выполненный в соответствии с шаблоном Business Delegate, может быть локальным представителем объекта, находящегося на бизнес-уровне.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Proxy" на стр. 483 Приложения А.

Поскольку в адресной книге хранятся все сведения о профессиональных и личных контактах пользователя, со временем ее размер выходит за оптимальные пределы. Кроме того, пользователю далеко не в каждом сеансе работы с PIM-приложением нужна адресная книга. По этим причинам необходимо создать некий промежуточный объект адресной книги, который должен быстро загружаться при запуске приложения. Для реализации этой задачи в данном примере используется шаблон Proxy, в соответствии с которым создается прокси-объект, представляющий реальную адресную книгу.

В листинге 3.31 представлен интерфейс `AddressBook`, с помощью которого обеспечивается доступ к адресной книге PIM-приложения. Этот интерфейс должен определять, как минимум, методы добавления новых контактов, а также получения и сохранения адресов.

Листинг 3.31. AddressBook.java

```
1. import java.io.IOException;
2. import java.util.AttayList;
3. public interface AddressBook {
4.   public void add(Address address);
5.   public ArrayList getAllAddresses();
6.   public Address getAddress(String description);
7.
8.    public void open();
9.    public void save();
10.)
```

Получение данных из адресной книги может потребовать весьма длительного времени (именно этим, по-видимому, объясняется повсеместная нелюбовь пользователей к адресным книгам). Поэтому прокси-объект должен оттягивать, насколько это возможно, момент создания реальной адресной книги. Конечно, вся ответственность за создание адресной книги, лежит именно на прокси-объекте AddressBookProxy (листинг 3.32), но выполнить эту операцию он должен только тогда, когда она действительно необходима.

Листинг 3.32. AddressBookProxy.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. public class AddressBookProxy implements AddressBook{
6.   private File file;
7.   private AddressBookImpl addressBook;
8.   private ArrayList localAddresses = new ArrayList();
9.
10.  public AddressBookProxy(String filename){
11.     file = new File(filename);
12.  }
13.
14.  public void open(){
15.     addressBook = new AddressBookImpl(file);
16.     Iteraror addressIterator = localAddresses.iterator();
17.     while (addressIterator.hasNext()){
18.       addressBook.add((Address)addressIterator.next());
19.     }
20.  }
21.
22.  public void saved(
23.     if (addressBook != null){
24.       addressBook.save();
25.     } else if (!localAddresses.isEmpty()){
26.       open();
27.       addressBook.save();
28.     }
29.  }
30.
31.  public ArrayList getAllAddresses() {
32.     if (addressBook == null) (
33.       open();
34.     }
35.     return addressBook.getAllAddresses();
36.  }
37.
38.  public Address getAddress(String description){
39.     if (!localAddresses.isEmpty()){
40.       Iterator addressIterator = localAddresses.iterator();
41.       while (addressIterator.hasNext()){
42.            AddressImpl address = (AddressImpl)addressIterator.next();
43.            if (address.getDescription().equalsIgnoreCase(description)){
44.                 return address;
45.            }
46.       )
47.     }
48.     if (addressBook == null){
49.       open();
50.     }
```

```
51.     return addressBook.getAddress(description);
52. }
53.
54. public void add(Address address))
55.     if (addressBook != null){
56.       addressBook.add(address);
57.     } else if (!localAddresses.contains(address)){
58.       localAddresses.add(address) ;
59.     }
60. }
61.}
```

Обратите внимание на то, что класс AddressBookProxy имеет собственную коллекцию ArrayList, предназначенную для хранения адресов. Это позволяет прокси-объекту в тех случаях, когда пользователь добавляет адрес с помощью метода add, использовать свою внутреннюю адресную книгу, не обращаясь к реальной адресной книге.

Реальная адресная книга в приложении представлена классом AddressBookImpl (листинг 3.33). Этот класс связан с файлом, в котором сохраняется содержимое коллекции ArrayList, представляющее собой все адреса, когда-либо внесенные пользователем в адресную книгу. Экземпляр класса AddressBookProxy создает объект класса AddressBookImpl только в том случае, когда это действительно необходимо — например, когда пользователь вызывает метод getAllAddresses.

Листинг 3.33. AddressBookImpl.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. public class AddressBookImpl implements AddressBook {
6.   private File file;
7.   private ArrayList addresses = new ArrayList ();
8.
9.   public AddressBookImpl(File newFile)  {
10.     file = newFile;
11.     open();
12. )
13.
14. public ArrayList getAllAddresses(){ return addresses; }
15.
16. public Address getAddress(String description)  {
17.     Iterator addressIterator = addresses.iterator();
18.     while (addressIterator.hasNext()){
19.       AddressImpl address =  (AddressImpl)addressIterator.next();
20.       if  (address.getDescription().equalsIgnoreCase(description)){
21.           return address;
22.       }
23.     }
24.     return null;
25. }
26.
27. public void add(Address address)  {
28.     if  (!addresses.contains(address)){
29.       addresses.add(address);
30.     )
31. }
32.
33. public void open(){
```

```
34.     addresses = (ArrayList)FileLoader.loadData(file);
35.  }
36.
37.  public void saved (
38.     FileLoader.storeData(file, addresses);
39.  >
40.}
```

СИСТЕМНЫЕ ШАБЛОНЫ

Глава 4

Введение

Системные шаблоны (system patterns) — это наиболее "пестрая" из четырех групп шаблонов. Они представляют приложение на архитектурном уровне — уровне наивысшей абстракции. Системные шаблоны могут применяться в приложении для осуществления большинства процессов и даже для поддержки взаимодействия разных приложений.

Каждый из рассмотренных в данной главе системных шаблонов имеет свое назначение.

- Model-View-Controller (MVC). Разделение компонента или подсистемы на три логические части (модель, представление и контроллер) с целью облегчения модификации или настройки каждой части в отдельности.
- Session. Обеспечение серверам распределенных систем возможности различения клиентов, что позволяет приложениям ассоциировать определенные состояния с клиент/серверными коммуникациями.
- Worker Thread. Улучшение пропускной способности и минимизация средней задержки.
- Callback. Обеспечение клиенту возможности регистрации на сервере для выполнения расширенных операций. Это позволяет серверу извещать клиента о завершении операции.
- Successive Update. Обеспечение клиенту возможности постоянного получения обновлений от сервера. Такие обновления обычно отражают изменения данных сервера, впервые появившиеся или обновленные ресурсы либо изменения в состоянии бизнес-модели.
- Router.Отделение источников информации от ее получателей.
- Transaction. Группирование коллекций методов таким образом, чтобы они либо были все успешно выполнены, либо все завершились неудачно.

Model-View-Controller (MVC)

Свойства шаблона

Тип: поведенческий
Уровень: компонент/архитектура

Назначение

Разделение компонента или подсистемы на три логические части (модель, представление и контроллер) с целью облегчения модификации или настройки каждой части в отдельности.

Представление

Предположим, разработчику нужно организовать в PIM-приложении работу с сущностью, которая должна представлять некое контактное лицо какой-то организации, в которой пользователь заинтересован в профессиональном плане. Для этого разработчик скорее всего создаст один класс, в котором будет храниться вся контактная информация: фамилия и имя контактного лица, название организации, занимаемая должность и т.п. Кроме того, в этот класс он включит программный код, предназначенный для визуального представления информации о контактном лице, например, ряд полей данных, собранных на панели. Наконец, разработчик позаботится о том, чтобы в этом классе присутствовали методы, которые бы обновляли информацию класса в соответствии с изменениями, вносимыми пользователем через графический интерфейс.

Но такой подход, который можно назвать традиционным, станет впоследствии причиной следующих проблем.

1. Хотя на первый взгляд все перечисленные выше составляющие класса вроде бы и предназначены для представления контактной информации и должны быть собраны в одном классе, все же можно заметить, что использование одного класса делает программный код более сложным и неудобным в сопровождении.
2. Весьма трудно предусмотреть развитие программного кода такого класса. Что делать, например, в том случае, если потребуется организовать различные формы представления контактной информации? Используя один класс, даже такую простую задачу решить довольно сложно.

Для того чтобы уйти от ненужной сложности и заранее подготовиться к будущим изменениям, лучше всего выделить три функциональные части одного класса в три разных класса. При таком подходе один класс представляет бизнес-информацию, второй занимается ее визуальным отображением, а третий обеспечивает соответствие между графическим интерфейсом и бизнес-информацией.

Таким образом, все три части сущности, представляющей в бизнес-модели контактное лицо, связаны друг с другом и работают, как единое целое. Если впоследствии придется вносить изменения в функции этой сущности, такие изменения скорее всего

затронут только один класс. Например, для того чтобы изменить представление контактной информации в соответствии с требованиями, выдвигаемыми к отображению данных в Web-броузере, достаточно будет лишь создать представление в виде HTML-страницы.

Этот шаблон, названный **Model-View-Controller** (MVC), очень полезен в тех случаях, когда нужно создавать компоненты, которые одновременно должны соответствовать требованиям гибкости и удобства сопровождения.

Обычно этот шаблон используется там, где заранее планируется внесение изменений и повторного использования кода, поскольку разделение сложного компонента на три класса или подсистемы требует дополнительных усилий на этапе проектирования архитектуры системы.

Областьприменения

Шаблон MVC полезен в тех случаях, когда компонент или подсистема имеет некоторые из следующих характеристик.

- Компонент или подсистему можно представлять несколькими различными способами. При этом внутреннее представление компонента или подсистемы в системе может полностью отличаться от представления на экране.
- Необходимо реализовывать несколько различных типов поведения. Иными словами, один и тот же компонент может инициализировать какие-то операции по запросам, поступающим из разных источников, и в зависимости от того, каков именно этот источник, поведение компонента может быть разным.
- Поведение или представление компонента изменяется в процессе его использования.
- Следует обеспечить возможность адаптации или повторного использования компонента в разных системах с минимальными изменениями в его программном коде.

Описание

Разработчики объектно-ориентированных систем постоянно сталкиваются с проблемой создания универсальных компонентов. Эта проблема проявляется особенно остро в тех случаях, когда компоненты должны быть по своей природе сложными или гибкими.

Рассмотрим, например, иакую таблицу. Концепцию таблицы можно применять самыми разными способами, т.е. в зависимости от потребностей приложения. Можно, например, подойти к реализации таблицы как к способу хранения данных в виде логической структуры, состоящей из ячеек, строк и столбцов. Однако этого не достаточно, поскольку существует много способов управления хранящимися данными и их **представлением.**

Например, если говорить о хранении, то таблицам может поддерживать лишь какие-то определенные формы данных (только десятичные числа) или же может разрешать выполнение специальных операций (таких, как суммирование). Она может вести себя, как таблица базы данных, в которой строки представляют отдельные за-

писи (группы элементов данных, представляющих одну сущность), а столбцы представляют поля (типы данных, одинаково идентифицируемых и одинаково хранящихся во всех записях). Может быть и совершенно иная ситуация, когда таблица не предъявляет никаких ограничений к хранящимся в ней данным, а также не требует как-то по-особому рассматривать ее строки и столбцы.

В приложении таблица может представляться также по-разному. Она может отображаться в виде сетки, графика или диаграммы. Одним из представлений может быть вообще отсутствие графического представления как такового. Кроме того, используя одни и те же нижележащие средства хранения данных, она может представлять информацию в нескольких различных формах, например, в виде сетки со связанной с ней диаграммой, обновление которой происходит при вводе значений пользователем в ячейки сетки.

Когда даже столь простой объект, как таблица, может использоваться столь большим количеством разных способов, разработчик сталкивается с дилеммой. С одной стороны, хорошо иметь возможность повторного использования программного кода, чтобы не создавать каждую новую таблицу с самого начала. Но с другой стороны, трудно понять, как разработать подобный универсальный компонент, чтобы он был повторно используемым. Слишком типовая реализация потребует значительной доработки при каждом повторном использовании, которая может свести на нет все преимущества этого повторного использования.

Шаблон MVC предлагает элегантную альтернативу. Согласно этому шаблону, сложный элемент определяется в терминах трех логических составляющих.

- Модель (model). Совокупность данных, отражающая состояние элемента, а также средства, обеспечивающие изменение состояния.
- Представление (view). Представление элемента (как визуальное, так и невизуальное).
- Контроллер (controller). Управляющая функциональность элемента, которая обеспечивает соответствие между операциями, выполняемыми с его представлением, и состоянием, образуемым моделью.

Применение шаблона MVC повсеместно можно встретить, например, в области управлением бизнеса. Действительно, руководители предприятий формируют модель, определяя цели и устанавливая правила, которые обеспечивают рост предприятия и выполнение его функций. Отделы продаж и маркетинга осуществляют представление модели, поскольку они представляют компанию и ее продукцию внешнему миру. Наконец, производственные отделы играют роль контроллера, так как они на основе информации, полученной от представления, предпринимают определенные шаги, направленные на изменение модели.

Подобное разделение элемента позволяет рассматривать каждую часть независимо от остальных (во всяком случае, практически независимо). Но для того чтобы элемент вел себя, как единое целое, необходимо, чтобы каждая часть имела соответствующий интерфейс с двумя другими. Так, например, представление должно иметь возможность отправлять сообщения контроллеру и получать информацию от модели, чтобы выполнять выдвигаемые к представлению требования. Однако шаблон MVC дает значительный выигрыш в том, что позволяет сравнительно легко заменять отдельные части компонента, обеспечивая тем самым чрезвычайно высокий уровень гибкости системы.

Например, таблица, реализованная в соответствии с шаблоном MVC, может изменяться из сетки в диаграмму путем простого изменения ее представления.

Рассмотренный пример с таблицей демонстрирует применение шаблона на уровне отдельного компонента, однако, ничто не препятствует использовать шаблон MVC на уровне архитектуры всей системы в целом. На уровне компонента модель управляет состоянием этого компонента, представление обеспечивает пользовательский интерфейс с ним, а контроллер выполняет обработку событий или привязку операций. На архитектурном уровне эти функции можно транслировать в подсистемы: модель представляет всю бизнес-модель, представление — отображение модели, а контроллер определяет бизнес-процессы.

Шаблон MVC относится к разряду тех шаблонов, которые "подталкивают" разработчика к интенсивному использованию инкапсуляции. В соответствии с принципами объектно-ориентированного программирования, рекомендуется определять элементы в терминах их интерфейса (т.е. как они взаимодействуют с внешним миром, другими объектами, компонентами или системами) и реализации (как они обеспечивают поддержание определенного состояния и как функционируют). Шаблон MVC поддерживает такой подход, поскольку в соответствии с ним элемент явно разделяется на три следующие части.

- Model. Реализация (состояние: атрибуты и внутреннее поведение).
- View. Внешний интерфейс (поведение: определение служб, которые могут использоваться для представления модели).
- Controller. Внутренний интерфейс (поведение: обслуживание запросов на обновление модели).

Реализация

Компонентная диаграмма шаблона MVC представлена на рис. 4.1.

Для описания данного шаблона используется компонентная диаграмма. Каждая из трех частей шаблона MVC представляет собой компонент, содержащий, в свою очередь, много классов и интерфейсов.

При реализации шаблона MVC обычно используются следующие компоненты.

- Model. Этот компонент содержит один или более классов и интерфейсов, отвечающих за обслуживание модели данных. Состояние модели сохраняется в атрибутах и реализации методов. Для того чтобы иметь возможность уведомлять компоненты-представления о любых изменениях, модель хранит ссылку на каждое зарегистрированное представление (одновременно может быть несколько представлений). Когда происходит изменение, каждый зарегистрированный компонент-представление извещается об этом изменении.
- View. Классы и интерфейсы представления обеспечивают презентацию данных, хранящихся в компоненте-модели. Представление может (но не обязано) состоять из компонентов визуального графического интерфейса. Для того чтобы представление могло получать извещения об изменении данных от модели, оно должно быть зарегистрировано в модели. Получая извещение об изменениях, компонент-представление принимает решениетом, нужно ли их отображать, и если нужно, то как именно это сделать.

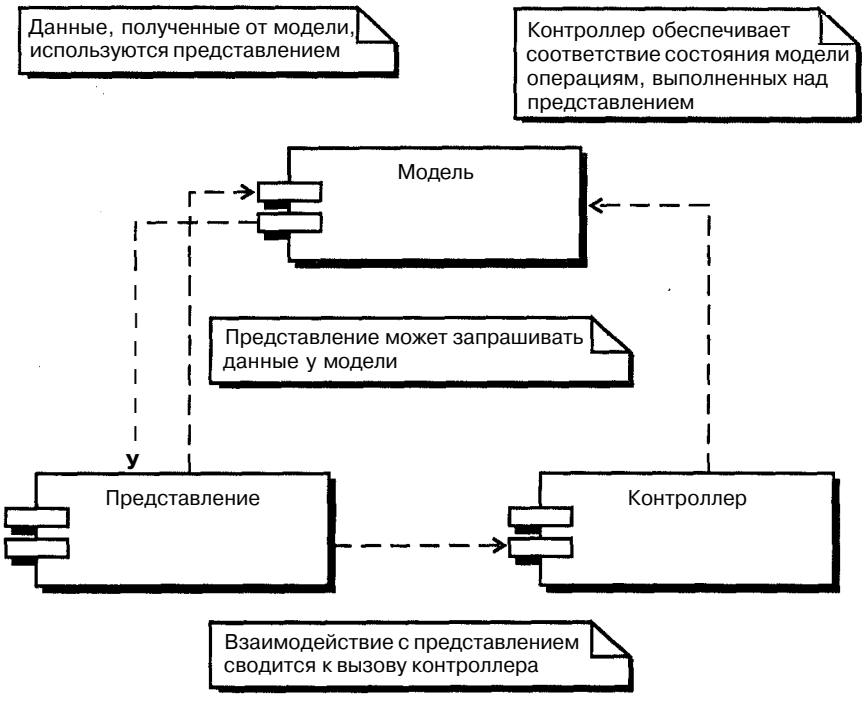


Рис. ***4.1.*** Компонентная диаграмма шаблона MVC

Компонент-представление также сохраняет ссылку на модель для получения данных от модели, но при этом представление может лишь получать данные без возможности их изменения. Представление может также использоваться для сокрытия контроллера, но при этом все запросы на изменение всегда пересылаются компоненту-контроллеру. В этой связи представление должно сохранять ссылку на один или несколько контроллеров.

- Controller. Этот компонент управляет изменениями модели. Он хранит ссылку на компонент-модель, который отвечает за осуществление изменений, и сам в свою очередь отвечает за вызов одного или нескольких методов обновления. Запрос на изменения может поступать от компонента-представления.

Достоинства и недостатки

Шаблон **MVC** предоставляет отличный способ создания гибких и адаптируемых к различным новым ситуациям элементов приложения. При этом гибкость может использоваться как статически, так и динамически. Под статической гибкостью подразумевается возможность добавления в приложение нового класса представления или контроллера, а под динамической — возможность замены объекта представления или контроллера во время работы приложения.

Обычно самая большая сложность при реализации шаблона MVC заключается в том, как правильно выбрать базовую презентацию элемента. Иными словами, нужно правильно разработать интерфейсы между моделью, представлением и контроллером. Элемент, выполненный в соответствии с шаблоном MVC, часто, как и большинство других программных объектов, должен удовлетворять какому-то определенному набору требований. Поэтому для реализации элемента таким образом, чтобы он не нес на себе отпечатка специфических особенностей приложения, необходимо иметь определенное видение и выполнить тщательный анализ.

Варианты

Варианты шаблона MVC чаще всего возникают вокруг различных реализаций представления.

- Рассылка информации моделью или ее получение представлением. Реализовать шаблон MVC можно одним из двух способов. В первом случае модель самостоятельно рассылает извещения об обновлениях своему представлению (или представлениям), а во втором— представление, по мере необходимости, запрашивает информацию у модели. Выбор конкретного способа влияет на реализацию взаимоотношений компонентов в системе.
- Несколько представлений. Модель может передавать информацию нескольким представлениям. Этот вариант особенно часто используется в некоторых реализациях графического интерфейса, поскольку в них одни и те же данные иногда должны представляться по-разному.
- Представления "смотри, но не трогай". Не всем представлениям нужен контроллер. Некоторые представления лишь отображают данные модели, но не поддерживают никаких средств, которые обеспечивали бы изменения в модели с помощью представления.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Observer (стр. 111). Шаблон MVC для управления коммуникациями часто пользуется шаблоном Observer. Обычно последний реализуется следующими частями системы.
  - Представлением и контроллером, чтобы изменения в представлении вызывали реакцию контроллера.
  - Моделью и представлением, чтобы **представление** извещалось обо всех изменениях в модели.
- Strategy (стр. 130). Контроллер часто реализуют на основе шаблона Strategy, что позволяет упростить процесс его замены другим контроллером.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Model-View-Controller (MVC)" на стр. 492 Приложения А.

В данном примере показана реализация шаблона MVC на компонентном уровне, обеспечивающая управление контактами в **PIM-приложении**. Класс `ContactModel` (листинг 4.1) в данном примере является моделью и содержит информацию об имени и фамилии контактного лица, а также об организации, которую оно представляет.

Листинг 4.1. `ContactModel.java`

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class ContactModel{
4.   private String firstName;
5.   private String lastName;
6.   private String title;
7.   private String organization;
8.   private ArrayList contactViews = new ArrayList ();
9.
10.  public ContactModel(){
11.     this(null);
12.  }
13.  public ContactModel (ContactView view){
14.     firstName = "";
15.     lastName = "";
16.     title = "";
17.     organization = "";
18.     if (view != null){
19.       contactViews.add(view);
20.     }
21.  }
22.
23.  public void addContactView(ContactView){
24.     if (!contactViews.contains(view)){
25.      contactViews.add(view) ;
26.     }
27.  }
28.
29.  public void removeContactView(ContactView view){
30.     contactViews.remove(view);
31.  }
32.
33.  public String getFirstName(){ return firstName; }
34.  public String getLastName(){ return lastName; }
35.   public String getTitle(){ return title; }
36.   public String getOrganization(){ return organization; }
37.
38.   public void setFirstName(String newFirstName){ firstName = newFirstName; }
39.   public void setLastName(String newLastName){ lastName = newLastName; }
40.   public void setTitle(String newTitle){ title = newTitle; }
41.   public void setOrganization(String newOrganization){ organization =
    newOrganization; }
```

```
42.
43.  public void updateModel(String newFirstName, String newLastName,
44.     String newTitle, String newOrganization){
45.    if (!isEmptyString(newFirstName) ){
46.      setFirstName(newFirstName) ;
47.    }
48.    if (!isEmptyString(newLastName)){
49.      setLastName(newLastName) ;
50.    }
51.    if (!isEmptyString(newTitle)){
52.       setTitle(newTitle);
53.    )
54.    if (!isEmptyString(newOrganization)){
55.       setOrganization(newOrganization);
56.    )
57.    updateView();
58. }
59.
60.  private boolean isEmptyString input){
61.    return ((input == null) || input.equals(""));
62. )
63.
64.  private void updateView(){
65.    iterator notifyViews = contactViews.iterator();
66.    while (notifyViews.hasNext()){
67.       ((ContactView)notifyViews.next()).refreshContactView(firstName,
   lastName, title, organization);
68.    }
69. }
70. }
```

Класс ContactModel содержит коллекцию ArrayList объектов класса ContactView (листинг 4.2) и обновляет последние при каждом изменении данных модели. Стандартное поведение всех представлений определяется методом refreshContactView интерфейса ContactView.

Листинг 4.2. ContactView.java

```
1. public interface ContactView{
2.  public void refreshContactView(String firstName,
3.    String lastName, String title, String organization);
4. }
```

В данном примере используется два представления. Первое, ContactDisplay-View (листинг 4.3), обеспечивает отображение обновленной информации модели, но не поддерживает работу с контроллером, являясь примером реализации поведения "только чтение".

Листинг 4.3. ContactDisplayView.java

```
1. import javax.swing.JPanel;
2. import javax.swing.JScrollPane;
3. import javax.swing.JTextArea;
4. import java.awt.BorderLayout;
5. public class ContactDisplayView extends JPanel implements ContactView{
6.  private JTextArea display;
7.
```

```
8.   public ContactDisplayView(){
9.      createGui();
10.  }
11.
12.  public void createGui(){
13.     setLayout(new BorderLayout());
14.     display = new JTextArealO, 40);
15.     display.setEditable(false);
16.     JScrollPane scrollDisplay = new JScrollPane(display);
17.     this.add(scrollDisplay, BorderLayout.CENTER);
18.  }
19.
20.  public void refreshContactView(String newFirstName,
21.     String newLastName, String newTitle, String newOrganization){
22.     display.setText("UPDATED CONTACT:\nNEW VALUES:\n" +
23.        "\tName: " + newFirstName + " " + newLastName +
24.        "\n" + "\tTitle: " + newTitle + "\n" +
25.        "\tOrganization: " + newOrganization);
26.  }
27.}
```

Второе представление, `ContactEditView` (листинг 4.4), позволяет пользователю обновить контактную информацию, определенную моделью.

Листинг 4.4. `ContactEditView.java`

```
1. import javax.swing.BoxLayout;
2. import javax.swing.JButton;
3. import javax.swing.JLabel;
4. import javax.swing.JTextField;
5. import javax.swing.JPanel;
6. import java.awt.GridLayout;
7. import java.awt.BorderLayout;
8. import java.awt.event.ActionListener;
9. import java.awt.event.ActionEvent;
10.public class ContactEditView extends JPanel implements ContactView{
11.  private static final String UPDATE_BUTON = "Update";
12.  private static final String EXIT_BUTON = "Exit";
13.  private static final String CONTACT_FIRST_NAME = "First Name ";
14.  private static final String CONTACT_LAST_NAME = "Last Name ";
15.  private static final String CONTACT_TITLE = "Title ";
16.  private static final String CONTACT_ORG = "Organization ";
17.  private static final int FNAME_COL_WIDTH = 25;
18.  private static final int LNAME_COL_WIDTH = 40;
19.  private static final int TITLE_COL_WIDTH = 25;
20.  private static final int ORG_COL_WIDTH = 40;
21.  private ContactEditController controller;
22.  private JLabel firstNameLabel, lastNameLabel, titleLabel/
   organizationLabel;
23.  private JTextField firstName, lastName, title, organization;
24.  private JButton update, exit;
25.
26.  public ContactEditView(ContactModel model){
27.     controller = new ContactEditController(model, this);
28.    createGui();
29. )
30.  public ContactEditView(ContactModel model, ContactEditController
   newController){
31.     controller = newController;
32.     createGui();
```

```
33. }
34.
35. public void createGui(){
36.    update = new JButton(UPDATE_BUTTON);
37.    exit = new JButton(EXIT_BUTTON);
38.
39.    firstNameLabel = new JLabel(CONTACT_FIRST_NAME);
40.    lastNameLabel(CONTACT_LAST_NAME);
41.    titleLabel = =new JLabel(CONTACT_TITLE);
42.    organizationLabel = new JLabel(CONTACT_ORG);
43.
44.    firstName = new JTextField(FNAME_COL_WIDTH);
45.    lastName = new JTextField(LNAME_COL_WIDTH);
46.    title = new JTextField(TITLE_COL_WIDTH);
47.    organization = new JTextField(ORG_COL_WIDTH);
48.
49.    JPanel editPanel = new JPanelO;
50.   editPanel.setLayout(new BoxLayout(editPanel, BoxLayout.X_AXIS)) ;
51.
52.    JPanel labelPanel = new JPanel();
53.    labelPanel.setLayout(new GridLayout(0, 1));
54.
55.    labelPanel.add(firstNameLabel) ;
56.    labelPanel.add(lastNameLabel) ;
57.    labelPanel.add(titleLabel);
58.    labelPanel.add(organizationLabel);
59.
60.   editPanel.add(labelPanel) ;
61.
62.    JPanel fieldPanel = new JPanel();
63.    fieldPanel.setLayout(new GridLayout(0, 1));
64.
65.    fieldPanel.add(firstName);
66.    fieldPanel.add(lastName);
67.   fieldPanel.add(title) ;
68.   fieldPanel.add(organization) ;
69.
70.   editPanel.add(fieldPanel) ;
71.
72.    JPanel controlPanel = new JPanel();
73.   controlPanel.add(update) ;
74.   controlPanel.add(exit) ;
75.   update.addActionListener(controller) ;
76.    exit.addActionListener(new ExitHandler());
77.
78.  setLayout(new BorderLayout() ) ;
79.    add(editPanel, BorderLayout.CENTER);
80.    add(controlPanel, BorderLayout.SOUTH);
81. }
82.
83. public Object getUpdateRef(){ return update; }
84. public String getFirstName(){ return firstName.getText(); }
85. public String getLastName(){ return lastName.getText(); }
86. public String getTitle(){ return title.getText (); }
87. public String getOrganization(){ return organization.getText(); }
88.
89. public void refreshContactView(String newFirstName,
90.    String newLastName, String newTitle,
91.    String newOrganization){
92.    firstName.setText(newFirstName);
93.    lastName.setText(newLastName);
94.   title.setText(newTitle) ;
95.    organization.setText(newOrganization);
```

```
96. }
97.
98. private class ExitHandler implements ActionListener{
99.    public void actionPerformed(ActionEvent event){
100.      System.exit(0);
101.   }
102. }
103. }
```

Обновление модели возможно благодаря наличию контроллера, связанного с классом ContactEditView. В данном примере управление взаимодействием между экземпляром класса ContactEditView и связанным с ним экземпляром класса ContactEditController (листинг 4.5) осуществляется на основе средств обработки событий языка Java (с применением расширения шаблона Observer). Объект класса ContactEditController обновляет объект класса ContactModel, когда процесс обновления инициируется объектом класса ContactEditView. Для обновления вызывается метод updateModel с новыми данными, полученными из редактируемых полей связанного с экземпляром класса ContactEditController представления.

Листинг 4.5. ContactEditController.java

```
1. import java.awt.event.*;
2.
3. public class ContactEditController implements ActionListener{
4.  private ContactModel model;
5.  private ContactEditView view;
6.
7.  public ContactEditController(ContactModel m, ContactEditView v){
8.     model = m;
9.     view = v;
10. }
11.
12.  public void actionPerformed(ActionEvent evt){
13.     Object source = evt.getSource();
14.     if (source == view.getUpdateRef()){
15.       updateModel();
16.     }
17. }
18.
19.  private void updateModel(){
20.     String firstName = null;
21.     String lastName = null;
22.     if (isAlphabetic(view.getFirstName())){
23.     firstName = view.getFirstName();
24. }
25.  if (isAlphabetic(view.getLastName())){
26.     lastName = view.getLastName();
27. )
28. model.updateModel( firstName, lastName,
29.     view.getTitle(), view.getOrganization());
30. }
31.
32.  private boolean isAlphabetic(String input){
33.    char [] testChars = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'};
34.     for (int i = 0; i < testChars.length; i++){
35.       if (input.indexOf(testChars[i]) != -1){
36.            return false;
37.       }
```

```
38.   }
39.       return true;
40.   }
41.}
```

Session

Свойства шаблона

Тип: обрабатывающий
Уровень: архитектурный

Назначение

Обеспечение серверам распределенных систем возможности различения клиентов, что позволяет приложениям ассоциировать определенные состояния с клиент/серверными коммуникациями.

Представление

Если PIM-приложение работает в сети, у разработчика возникает естественное желание централизовать хранение некоторой информации, например, сведений о потребителях компании.

По-видимому, при таком подходе клиентам понадобится постоянно обновлять контактную информацию на сервере, причем обновления могут поступать в разной последовательности. Пользователи, например, могут сначала модифицировать информацию о должности контактного лица и названии компании, которую оно представляет, а спустя некоторое время — информацию об адресе. Поскольку операции обновления информации об адресе могут выполняться несколькими пользователями одновременно и в режиме реального времени, необходимо каким-то образом обеспечить клиенту возможность взаимодействовать с сервером, который в этот момент обменивается подобными данными с другими клиентами.

Итак, перед нами проблема: каким образом можно отследить все изменения, вносимые каким-то из пользователей и относящиеся к определенной контактной информации, когда эти изменения выполняются в произвольном порядке? Многочисленные клиенты будут обновлять информацию одновременно, поэтому сервер должен однозначно определять, какие изменения поступили от какого клиента. В противном случае клиенты могут по ошибке обновить информацию не тех потребителей, с которыми они работают.

Самым эффективным способом решения этой проблемы является связь с каждым пользователем некоторого временного идентификатора, что позволяет серверу отслеживать все операции, выполняемые пользователем с данным идентификатором. Когда пользователь начинает редактировать информацию, хранящуюся на сервере, сервер открывает *сеанс* (session), присваивая ему *идентификатор сеанса* (session ID). При выполнении пользователем очередной операции, такой как добавление или уда-

ление адреса, приложение пользователя отправляет идентификатор сеанса серверу. При обновлении контактной информации экземпляр приложения, с которым работает пользователь, отправляет серверу *завершающее сообщение* (finalize message), которое говорит о том, что клиент завершил обновление контактной информации. Получив такое сообщение, сервер закрывает сеанс.

Такое решение, формализованное в виде шаблона Session, предоставляет серверу массу преимуществ. Использование этого шаблона дает серверу возможность различать клиентов и отслеживать операции, выполняемые каждым конкретным клиентом. Кроме того, этот шаблон позволяет серверу сохранять динамически изменяющуюся информацию, не прибегая к ее хранению на машине клиента. Применение сеансов позволяет серверу кэшировать пользовательскую информацию в памяти до тех пор, пока пользователь не завершит ее редактирование.

Область применения

Шаблон Session можно применять, как в клиент/серверной, так и в одноранговой сетевой среде, в которой выполняются следующие требования.

- Идентификация клиента. Реализован способ различения клиентов многопользовательской системы.

Кроме того, шаблон Session обычно используется в системах, которым присуща хотя бы одна из двух следующих характеристик.

- Непрерывность операций. Имеется возможность связывать определенные операции, выполняющиеся в системе, с другими подобными операциями. Такие операции могут осуществляться в соответствии с транзакционной (transactional) или потоковой (workflow) моделями.
- Целостность данных. Обеспечивается связь данных с клиентом на протяжении всего времени, в течение которого клиент взаимодействует с сервером.

Описание

Информация о шаблоне Session излагается в приведенных ниже подразделах.

Коммуникации, ориентированные на состояние, и коммуникации, не ориентированные на состояние

Коммуникации в распределенных системах могут быть как *ориентированными на состояние* (stateful), так и *неориентированными на состояние* (stateless).

- Примером коммуникаций, ориентированных на состояние, являются сокеты (socket). При использовании сокетов можно обеспечить, чтобы запросы, отправляемые от клиентов к серверу, были последовательными, а также чтобы сервер отслеживал предыдущие запросы.
- В тех случаях, когда сервер не следит за тем, что происходило раньше, имеют место коммуникации, не ориентированные на состояние. В этом случае сервер не отличает один запрос от другого, в связи с чем каждый запрос должен быть

"вещью в себе", т.е. вся информация, которая нужна для обеспечения ответа на запрос, должна находиться в самом запросе. Эта модель, как правило, очень проста в реализации и позволяет значительно упростить как программный код клиента, так и сервера.

Примером реализации модели коммуникаций, не ориентированных на состояние, является World Wide Web (WWW). Основа WWW, протокол Hypertext Transfer Protocol (HTTP), – это не ориентированный на состояние механизм обмена информацией между Web-броузером и сервером. Протокол представляет собой небольшую группу простых операций и хорошо справляется со своим главным назначением — обеспечением передачи документов через Web.

Приложениям часто нужны коммуникации, ориентированные на состояние

Иногда приложениям нужны более сложные коммуникационные механизмы, чем коммуникации, не ориентированные на состояние. В частности, бизнес-приложения нередко требуют наличия поддержки некоторых из следующих форм обмена информацией или даже всех этих форм одновременно.

- Поток (workflow). Направленная последовательность бизнес-операций.
- Транзакция (transaction). Набор взаимосвязанных операций, которые рассматриваются как единое целое.
- Прикладные данные (application data). Информация, связанная с процессом взаимодействия клиента и сервера.

Рассмотрим классическое приложение электронной коммерции. Пока потребитель выбирает требуемые ему продукты, система должна хранить данные, которые представляют содержимое потребительской корзины. Кроме того, при покупке через Internet используются потоки операций, с помощью которых определяются последовательности действий, необходимых для выписки счета, оплаты товаров и доставки заказа. Понятно, что таким приложениям требуется каким-то образом представлять все операции, происходящие между клиентом и сервером во время посещения электронного магазина.

Шаблон Session и коммуникации, ориентированные на состояние

В таких довольно сложных приложениях шаблон Session может оказаться очень полезным. С помощью этого шаблона можно идентифицировать множество клиентов, а также создавать один или несколько объектов, представляющих состояние обмена информацией между определенным клиентом и сервером. Шаблон позволяет обеспечить непрерывность операций, выполняемых клиентом на сервере на протяжении определенного периода времени, а также разносить их по времени.

Особенно полезен шаблон Session в тех случаях, когда нужно обеспечить поддержку потока операций, которыми с сервером обменивается множество клиентов. Не используя концепции сеанса, сервер не смог бы эффективно отслеживать, какие операции принадлежат какому клиенту.

В Web-приложениях часто можно увидеть, как сеансы вводятся исключительно для этих целей. В обычных условиях Web работает в соответствии с моделью коммуникаций, не ориентированных на состояние. Однако в тех случаях, когда нужно обеспечить поддержку электронной коммерции, требуется реализовать какой-то способ управления сеансами. Когда пользователи совершают покупки на Web-узле, прежде чем оформить заказ, они обычно добавляют в потребительскую корзину и удаляют из нее множество товаров. Если для отслеживания этой деятельности и хранения сведений обо всех товарах, которые могут приобрести пользователи, не использовать сеансы, Web-узел электронной коммерции будет сведен к одной простой форме заказа товара через Internet.

Коммуникации, ориентированные на состояние, в реальном мире

Любая ситуация в реальной жизни, в которой используются концепции идентификации и транзакционного состояния, является примером реализации шаблона Session. Например, официант в закусочной использует при идентификации клиентов (посетителей) и обслуживании метод визуального распознавания лиц. Это позволяет серверу (официанту) различать запросы, поступающие от разных посетителей, и справляться с несколькими запросами одновременно. Если, например, посетитель 42 поразмыслит как следует, он может заменить заказанную вместе с анчоусами пастраму на отбивную и добавить к заказу бутерброд с ветчиной.

Хотя посетитель сделал много запросов, сервер знает, что окончательный заказ нужно отнести к тому же посетителю и что ни одному из других посетителей не понадобится пастрама посетителя 42. Единственной проблемой этой системы может стать потеря другими посетителями терпения, пока посетитель 42 формирует свой заказ. Конечно, такая ситуации — это хороший повод для владельца заведения подумать над привлечением дополнительных работников, а также над тем, как сделать официантов многопоточными.

Реализация

При реализации шаблон Session должен соответствовать двум фундаментальным требованиям.

- Идентификация сеанса. Сервер должен иметь возможность сохранять идентификатор клиента на протяжении всей работы приложения.
- Представление сеанса. Для представления состояния сеанса может использоваться один или несколько объектов сеанса, в зависимости от потребностей приложения.

Достоинства и недостатки

Основное достоинство шаблона Session явственно следует из его характеристик: идентификация сущностей, запрашивающих обслуживание, и обеспечение работы с ресурсами по состоянию. Можно отметить и другие достоинства, которые могут про-

являться в зависимости от выбранной модели реализации шаблона. Например, если идентификация клиента выполняется на основе прохождения процедуры входа в систему, шаблон Session может при осуществлении доступа к ресурсам сервера управлять ведением учета и приоритетами. Если информация о сеансах хранится в базе данных, сервер может поддерживать ведение информации о состоянии клиента на протяжении целого ряда транзакций.

Недостатком шаблона Session является увеличение нагрузки на сервер, а также повышение сложности программного обеспечения серверной части. Помимо обычных задач, сервер, выполненный в соответствии с шаблоном Session, должен каким-то образом обеспечить идентификацию клиентов, сохранение и восстановление соответствующей информации, а также проверку идентификаторов клиентов, которая может выполняться в процессе работы приложения не один раз.

Варианты

Основные варианты реализации шаблона Session базируются на различных способах управления идентификацией и состоянием сеансов.

- *Управление идентификацией сеансов.* Здесь можно использовать три разных подхода.
  - Идентификация, основанная на обеспечении безопасности. Идентификатор сеанса для клиента образуется в результате прохождения процедуры входа (login).
  - Неявная идентификация. Автоматическая идентификация клиента обеспечивается на основе существования долговременного подключения к серверу.
  - Случайная идентификация. Сервер присваивает каждому клиенту некий уникальный идентификатор сеанса. Этот идентификатор выбирается случайным образом и используется лишь для отслеживания операций, выполняемых клиентом, в ходе текущего сеанса работы с сервером.
- *Управление состоянием сеансов.* В тех случаях, когда нужно управлять состоянием сеанса, можно сохранять соответствующую информацию либо на стороне клиента, либо на стороне сервера.
  - Объектно-ориентированное хранение на стороне клиента. Ответственность за хранение данных и, при необходимости, их отправку серверу возлагается на клиента. Это снижает общий уровень безопасности приложения, поскольку данные хранятся на клиентской машине, которая скорее всего защищена менее надежно, чем сервер. С другой стороны, данный подход упрощает связь данных с клиентом, так как сам клиент хранит информацию и отправляет ее серверу. Другим преимуществом данного подхода является то, что он позволяет снизить нагрузку сервера, возлагая на клиентское приложение ответственность за хранение своих данных.

    Способ реализации этого подхода зависит от выбранной технологии. Например, в HTTP данный подход реализован в виде файлов cookies.

- Объектно-ориентированное хранение на стороне сервера. Все данные клиентов сохраняются на сервере, который обрабатывает их в соответствии с запросами клиентов. Сервер обслуживает все данные приложений, что ведет к возрастанию нагрузки на сервер. Однако при этом обеспечивается более высокий уровень безопасности всей системы в целом, так как данные хранятся на хорошо защищенном сервере. Эффективность системы, выполненной в соответствии с таким подходом, также, как правило, повышается, поскольку в ней отсутствуют лишние операции пересылки данных. Проблема, которая может возникнуть при хранении данных на сервере, состоит в затруднениях с идентификацией клиентов, поскольку в приложении клиент отделен от принадлежащих ему данных.

  В HTTP и Java данный подход выражается в использовании класса `HttpSession`.

Родственные шаблоны

Отсутствуют.

Пример

Примечание

Полный работающий код данного примера совспомогательными классами, а также классом `RunPattern` приведен в разделе "Session" настр. 498 Приложения А.

На рис. 4.2 приведена компонентная диаграмма сеанса, ориентированного на клиента.

Вторая диаграмма, представленная на рис. 4.8, соответствует сеансу, обеспечиваемому сервером.

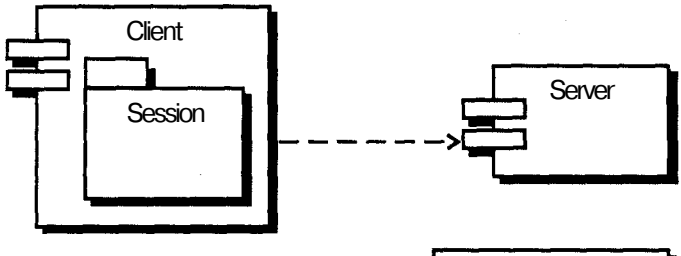


Рис. 4.2. Компонентная диаграмма сеанса, ориентированного на клиента

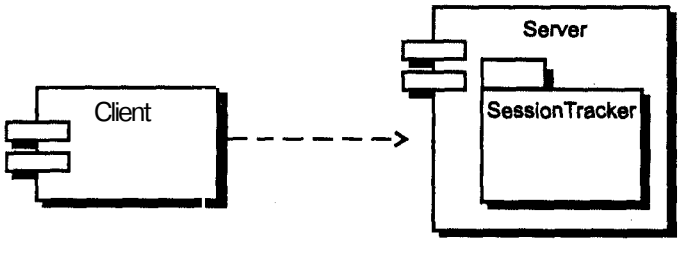


Рис. 4.3. Компонентная диаграмма сеанса, обеспечиваемого сервером

Диаграмма классов компонента сеанса представлена на рис. 4.4.

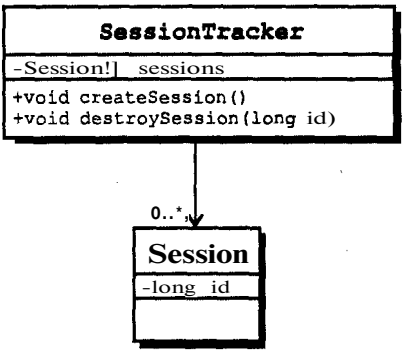


Рис. *4.4.* Диаграмма классов компонента сеанса

В данном примере клиент обращается к серверу с запросами на выполнение ряда операций по обновлению контактной информации в совместно используемой адресной книге. Пользователь может выполнять четыре операции.

- Добавление контакта.
- Добавление адреса (связан с текущим контактом).
- Удаление адреса (связан с текущим контактом).
- Сохранение изменений контакта и его адреса.

Эти операции определяются в классе SessionClient (листинг 4.6).

Листинг 4.6. SessionClient.java

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. public class SessionClient{
6.   private static final String SESSION_SERVER_SERVICE_NAME = "sessionServer";
7.   private static final String SESSION_SERVER_MACHINE_NAME = "localhost";
8.   private long sessionlD;
9.   private SessionServer sessionServer;
10.
11.  public SessionClient(){
12.     try{
13.       . String url = "//" + SESSION SERVER MACHINE NAME + "/" +
14.SESSION_SERVER_SERVICE_NAME;
15.       sessionServer = (SessionServer)Naming.lookup(url);
16.     }
17.     catch (RemoteException exc){}
18.     catch (NotBoundException exc){}
19.     catch (MalformedURLException exc){}
20.     catch (ClassCastException exc)()
21.  }
22.
23.  public void addContact(Contact contact) throws SessionException{
24.     try{
25.       sessionlD = sessionServer.addContact(contact, 0);
```

```
26.     }
27.     catch (RemoteException exc){}
28. }
29.
30. public void addAddress(Address address) throws SessionException{
31.     try{
32.       sessionServer.addAddress(address, sessionID);
33.     }
34.     catch (RemoteException exc){}
35. }
36.
37. public void removeAddress (Address address) throws SessionException{
38.     try{
39.       sessionServer.removeAddress(address, sessionID);
40.     }
41.     catch (RemoteException exc){}
42. }
43. public void commitChanges() throws SessionException{
44.     try(
45.       sessionID = sessionServer.finalizeContact(sessionID);
46.     }
47.     catch (RemoteException exc){}
48. }
49.}
```

Каждый метод клиента вызывает соответствующий метод удаленного сервера. Интерфейс SessionServer (листинг 4.7) определяет четыре метода, доступные клиентам через механизм RMI.

Листинг 4.7. SessionServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface SessionServer extends Remote{
4.   public long addContact(Contact contact, long sessionID) throws
   RemoteException, SessionException;
5.   public long addAddress(Address address, long sessionID) throws
   RemoteException, SessionException;
6.   public long removeAddress(Address address, long sessionID) throws
   RemoteException, SessionException;
7.   public long finalizeContact(long sessionID) throws RemoteException,
   SessionException;
8. }
```

Класс SessionServerImpl (листинг 4.8) реализует интерфейс SessionServer, обеспечивая тем самым сервер RMI. Он делегирует определение бизнес-логики классу SessionServerDelegate (листинг 4.9).

Листинг 4.8. SessionServerImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. public class SessionServerImpl implements SessionServer{
4.   private static final String SESSION_SERVER_SERVICE_NAME = "SessionServer";
5.   public SessionServerImpl(){
6.     try{
7.       UnicastRemoteObject.exportObject(this);
```

```
8.       Naming.rebind(SESSION_SERVER_SERVICE_NAME, this);
9.     }
10.    catch (Exception exc){
11.      System.err.println("Error using RMI to register the SessionServerImpl"
   + exc);
12.    }
13. }
14.
15. public long addContact(Contact contact, long sessionID) throws
   SessionException{
16.    return SessionServerDelegate.addContact(contact, sessionID);
17. }
18.
19. public long addAddress(Address address, long sessionID) throws
   SessionException{
20.    return SessionServerDelegate.addAddress(address, sessionID);
21. }
22.
23. public long removeAddress(Address address, long sessionID) throws
   SessionException{
24.    return SessionServerDelegate.removeAddress(address, sessionID);
25. }
26.
27. public long finalizeContact(long sessionID) throws SessionException{
28.    return SessionServerDelegate.finalizeContact(sessionID);
29. }
30.}
```

Листинг 4.9. SessionServerDelegate.java

```
1. import java.util.ArrayList;
2. import java.util.HashMap;
3. public class SessionServerDelegate{
4.   private static final long NO_SESSION_ID = 0;
5.   private static long nextSessionID = 1;
6.   private static ArrayList contacts = new ArrayList();
7.   private static ArrayList addresses = new ArrayList();
8.   private static HashMap editContacts = new HashMap ();
9.
10. public static long addContact(Contact contact, long sessionID) throws
   SessionException{
11.    if (sessionID <= NO_SESSION_ID){
12.      sessionID = getSessionID();
13.    }
14.    if (contacts.indexOf(contact) != -1){
15.      if (!editContacts.containsValue(contact) ){
16.          editContacts.put(new Long(sessionID), contact);
17.      }
18.      else{
19.      throw new SessionException("This contact is currently being edited by
   another user.",
20.            SessionException.CONTACT_BEING_EDITED);
21.      }
22.    }
23.    else{
24.      contacts.add(contact);
25.      editContacts.put(new Long(sessionID), contact;
26.    }
27.    return sessionID;
28. }
29.
```

```
30. public static long addAddress(Address address, long sessionID) throws
    SessionException{
31.     if (sessionID <= NO_SESSION_ID){
32.       throw new SessionException("A valid session ID is required to add an
    address",
33.             SessionException.SESSION_ID_REQUIRED);
34.     }
35.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
36.     if (contact == null){
37.       throw new SessionException("You must select a contact before adding an
    address",
38.             SessionException.CONTACT_SELECT_REQUIRED);
39.     }
40.     if (addresses.indexOf(address) == -1){
41.       addresses.add(address) ;
42.     }
43.     contact.addAddress(address) ;
44.     return sessionID;
45. }
46.
47. public static long removeAddress(Address address, long sessionID) throws
    SessionException{
48.     if (sessionID <= NO_SESSION_ID){
49.       throw new SessionException("A valid session ID is required to remove
    an address",
50.             SessionException.SESSION_ID_REQUIRED);
51.     }
52.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
53.     if (contact == null){
54.       throw new SessionException("You must select a contact before removing
    an address",
55.             SessionException.CONTACT_SELECT_REQUIRED);
56.     }
57.     if (address.indexOf(address) == -1){
58.       throw new SessionException("There is no record of this address",
59.           SessionException.ADDRESS_DOES_NOT_EXIST) ;
60.     }
61.     contact.removeAddress(address) ;
62.     return sessionID;
63. }
64.
65. public static long finalizeContact(long sessionID) throws SessionException{
66.     if (sessionID <= NO_SESSION_ID){
67.       throw new SessionException("A valid session ID is required to finalize
    a contact",
68.             SessionException.SESSION_ID_REQUIRED);
69.     }
70.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
71.     if (contact == null){
72.       throw new SessionException("You must select and edit a contact before
    committing changes",
73.             SessionException.CONTACT_SELECT_REQUIRED);
74.     }
75.     editContacts.remove(new Long(sessionID));
76.     return NO_SESSION_ID;
77. )
78.
79. private static long getSessionID(){
80.     return nextSessionID++;
81. }
82.
83. public static ArrayList getContacts (){ return contacts; }
84. public static ArrayList getAddresses(){ return addresses; }
```

```
85.  public static ArrayList getEditContacts(){ return new
   ArrayList(editContacts.values());}
86.}
```

Класс SessionServerDelegate генерирует идентификатор сеанса для клиентов, когда они выполняют свою первую операцию, — добавление объекта класса Contact. Для выполнения последующих операций с адресом контакта требуется наличие идентификатора сеанса, поскольку этот идентификатор используется в классе SessionServerDelegate для ассоциации адреса с конкретным объектом класса Contact.

Worker Thread

Также известен как Background Thread, Thread Pool

Свойства шаблона

Тип: обрабатывающий
Уровень: архитектурный

Назначение

Улучшение пропускной способности и минимизация средней задержки.

Представление

Применение в приложении многопоточности, как правило, вызывается желанием избавиться от "узких мест" с точки зрения производительности. Однако для того, чтобы реализовать средства поддержки многопоточности корректно, необходимо обладать определенными навыками. Один из способов максимизации эффективности многопоточного приложения состоит в использовании того факта, что не все выполненные в виде потоков задачи приложения имеют одинаковый приоритет. Для некоторых задач на первом месте стоит время выполнения. Другие же просто должны быть выполнены, а когда именно произойдет это выполнение — не столь важно.

Для того чтобы не расходовать свое время понапрасну, разработчик может отделить подобные задачи от приложения и воспользоваться шаблоном Worker Thread. Созданный в соответствии с этим шаблоном обработчик потока будет выбирать из очереди задачи и выполнять их в отдельном потоке. По окончании очередной задачи обработчик выбирает из очереди следующую задачу и все повторяется сначала.

Создание многопоточного приложения при использовании шаблона Worker Thread значительно упрощается, поскольку в тех случаях, когда не важно, как скоро будет выполнена задача, разработчику достаточно просто поместить ее в очередь, а все остальное сделает обработчик потока. Программный код такого приложения также упрощается, так как все объекты, работающие с потоками, скрыты внутри обработчика потока и очереди.

Область применения

Шаблон Worker Thread рекомендуется использовать в тех случаях, когда:

- нужно повысить пропускную способность приложения;
- необходимо обеспечить одновременное выполнение разных фрагментов кода.

Описание

Для реализации потоков в приложении можно воспользоваться традиционным подходом: при запуске новой задачи нужно создать новый объект Thread и запустить его на выполнение. Поток, представленный этим объектом, выполнит всю порученную ему работу и автоматически завершится. Как видно, все довольно просто. Однако создание экземпляра потока— это весьма расточительный процесс с точки зрения производительности, он требует немало времени и позволяет выполнить только одну задачу. Более эффективный способ заключается в создании объекта-"долгожителя" — специального обработчика потока, который будет выполнять одну задачу за другой.

В выполнении такой работы и состоит сущность шаблона Worker Thread. Обработчик потока, реализованный в соответствии с этим шаблоном, выполняет одну за другой множество не связанных друг с другом задач. Теперь уже не нужно создавать новый поток при каждом запуске новой задачи — достаточно лишь передать задачу уже существующему обработчику потока, который позаботится об остальном.

Однако возможна ситуация, когда обработчик потока занят выполнением очередной задачи, а приложение уже подготовило следующую задачу. В такой ситуации можно предложить одно из следующих решений.

- Приложение ждет до тех пор, пока обработчик потока не освободится от текущей задачи. Это решение очевидно, но оно практически сводит на нет все преимущества, предоставляемые многопоточностью.
- Приложение создает новый экземпляр обработчика потока всякий раз, когда текущий обработчик потока недоступен. Такое решение также лежит на поверхности, но оно, по сути, является возвратом к традиционной технологии, так как возможны ситуации, при которых для каждой новой задачи будет создаваться отдельный поток.

Наилучшее же решение проблемы временно недоступного обработчика потока состоит в том, чтобы сохранить задачи в очереди до тех пор, пока обработчик потока не освободится. Приложение помещает каждую новую задачу в очередь, а обработчик потока, закончив выполнение очередной задачи, проверяет, есть ли в очереди новые задачи, и если таковые имеются, запускает следующую задачу на выполнение. Конечно, это не дает преимущества относительно скорости выполнения задач, но во всяком случае освобождает приложение от необходимости ожидания, пока освободится обработчик потока. Если же нет задач для выполнения, обработчик периодически проверяет очередь. Помещение задачи в очередь — это в любом случае гораздо менее требовательный с точки зрения производительности процесс, чем создание нового потока.

Реализация

Диаграмма классов шаблона Worker Thread представлена на рис. 4.5.

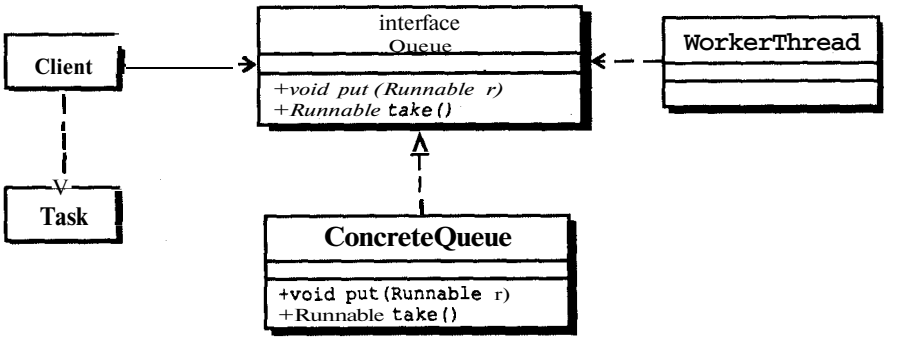


Рис. 4.5. Диаграмма классов шаблона Worker Thread

При реализации шаблона Worker Thread обычно используются следующие классы.

- Client. Клиент отвечает за создание экземпляров класса Task и помещение их в очередь, представленную экземпляром класса Queue.
- Task. Класс, содержащий задание которое нужно выполнить. Реализует интерфейс `java.lang.Runnable`.
- Queue. Интерфейс, определяющий методы, с помощью которых экземпляр класса Client помещает в очередь объекты класса Task, а экземпляр класса WorkerThread извлекает их из очереди.
- ConcreteQueue. Класс ConcreteQueue реализует интерфейс Queue и отвечает за сохранение в очереди задач и извлечение их из очереди. Он определяет порядок, в котором экземпляры класса `Task` поступают для выполнения к экземпляру класса WorkerThread.
- WorkerThread. Экземпляр этого класса извлекает задачи из очереди и выполняет их. Если в очереди нет задач, экземпляр класса WorkerThread ожидает, пока в очереди не появится новая задача. Поскольку классы Queue и WorkerThread тесно взаимосвязаны, часто класс WorkerThread выполняется в виде внутреннего класса в классе ConcreteQueue.

Достоинства и недостатки

Шаблон Worker Thread оказывает влияние на производительность в нескольких направлениях.

1. Клиенту для запуска различных заданий не нужно создавать несколько объектов потоков. Все, что он должен сделать, — это поместить задачу в очередь, что с точки зрения производительности требует значительно меньших накладных расходов, чем создание объекта потока.

2. Существующий, но не выполняющийся поток, все равно снижает производительность, так как планировщик выделяет часть машинного времени для выполнения потока, находящегося в состоянии готовности для выполнения. Создание и запуск потока для каждой задачи означает, что планировщик должен выделять ресурсы каждому такому потоку индивидуально. В сумме потери времени на такое планирование значительно больше, чем потери, которые возникают при наличии постоянно работающего обработчика потока. Иными словами, чем больше потоков, тем выше накладные расходы на их планирование. Если же задание находится в очереди и, соответственно, не выполняется, оно вообще не расходует машинного времени.
3. Данное решение имеет один недостаток, который может проявляться в тех случаях, когда задачи являются взаимозависимыми. Если очередь последовательна, возникновение подобной ситуации может привести к блокировке системы. Это, конечно же, весьма неприятно и с точки зрения многопоточности, и с точки зрения производительности.

   Для решения этой дилеммы можно предложить несколько подходов.

   - Создается столько обработчиков потоков, сколько задач необходимо выполнять одновременно. Это означает, что в приложении нужно организовать расширяемый пул потоков. Подробнее о пуле потоков говорится далее в подразделе "Варианты" данного раздела.
   - В очередь разрешается помещать только те задачи, которые не зависят от других задач. Иногда такой подход трудно реализовать. В таких случаях клиент должен не помещать задачу в очередь, а создать экземпляр собственного потока или запустить отдельную очередь с обработчиком потока.
   - Создается интеллектуальная очередь, которая может установить, какие задачи работают совместно, и принять решение, когда ту или иную задачу передать обработчику потока. Этот подход нужно применять только тогда, когда не осталось других возможностей, поскольку такая интеллектуальная очередь должна быть тесно связана с приложением, а сопровождение ее программного кода может превратиться в сплошной кошмар.

Варианты

Одним из вариантов реализации данного шаблона является пул потоков (thread pool). В этом случае в приложении существует не один, а несколько экземпляров обработчика потоков, образующих пул (отсюда и название). Этот пул управляет экземплярами класса WorkerThread. Пул создает обработчики потоков, определяет, нужны ли они еще, а также удаляет ставшие ненужными экземпляры обработчиков потоков.

Пул также решает, сколько экземпляров обработчиков нужно создать в момент запуска приложения, а также какое максимальное количество экземпляров может присутствовать в системе одновременно. Пул может либо создать несколько обработчиков потоков при запуске, что позволит всегда иметь несколько потоков, готовых к применению, либо ждать до тех пор, пока не возникнет необходимость в обработчике потока (отложенное создание экземпляров).

Однако и это решение не позволяет справиться с ситуацией, когда существует слишком много задач, ожидающих потока, а количество потоков ограничено. Система в таких случаях "засоряется", как водосточная труба. Для устранения этой проблемы можно предложить несколько решений.

- Увеличение количества обработчиков потоков. Это решение является ограниченным, поскольку оно направлено на устранение симптомов, а не самой проблемы. Обычно рекомендуется использовать более удачное решение.
- Снятие ограничений на количество задач в очереди. Данное решение позволяет очереди расти до тех пор, пока не будет исчерпана выделенная приложению память. Это решение лучше, чем увеличение количества обработчиков потоков, но также является ограниченным, поскольку физические ресурсы системы не могут быть бесконечными.
- Ограничение размера очереди. Когда количество задач, ждущих обработки, становится слишком большим, клиенты лишаются возможности вызывать методы, добавляющие в очередь новые задачи. Это позволяет разгрузить очередь и заняться обработкой ожидающих в ней задач.
- Отправка запроса клиентам, предписывающего приостановить постановку задач в очередь. Клиенты, получив такой запрос, могут либо приостановить отправку запросов, либо уменьшить их количество.
- Сброс запросов, стоящих в очереди. Если пулу известно, что клиенты смогут повторить необработанные запросы, он может просто сбрасывать новые запросы. Можно также сбрасывать старые запросы в тех случаях, когда высока вероятность того, что клиенты, которые их разместили в очереди, уже отключились.
- Разрешение клиентам запускать задачи самостоятельно. В этом случае клиент переходит в однопоточный режим и, таким образом, не может создать новую задачу до тех пор, пока не завершит предыдущую задачу.

Родственные шаблоны

Отсутствуют

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Worker Thread" на стр. 507 Приложения ***A.***

В типичном приложении нужно выполнить какую-то определенную работу. Иногда даже не важно, что это за работа, главное, чтобы она выполнялась. Это можно сравнить с уборкой дома. Если кто-то занялся в эту субботу уборкой, то уже не важно, что он делает в данный конкретный момент времени, лишь бы он это делал.

В данном примере для хранения задач используется очередь, представленная интерфейсом Queue (листинг 4.10). Интерфейс Queue определяет два базовых метода: put и take. Эти методы используются для добавления в очередь и удаления из нее задач, представленных интерфейсом RunnableTask (листинг 4.11).

Листинг 4.10. Queue.java

```
1. public interface Queue{
2.   void put(RunnableTask r) ;
3.    RunnableTask take();
4. }
```

Листинг 4.11. RunnableTask.java

```
1. public interface RunnableTask(
2.   public void executed;
3. }
```

Класс ConcreteQueue (листинг 4.12) реализует интерфейс Queue и создает обработчик потока, который выполняет объекты, реализующие интерфейс RunnableTask. Класс Worker, который является внутренним классом класса ConcreteQueue, имеет метод run, предназначенный для периодической проверки очереди на наличие в ней готовых к выполнению задач. Когда такая задача появляется, обработчик потока извлекает ее из очереди и запускает ее метод execute.

Листинг 4.12. ConcreteQueue.java

```
1. import java.util.Vector;
2. public class ConcreteQueue implements queue{
3.   private Vector tasks = new Vector() ;
4.   private boolean waiting;
5.   private boolean shutdown;
6.
7.   public void setShutdown(boolean isShutdown){ shutdown = isShutdown; }
8.
9.   public ConcreteQueue (){
10.    tasks = new Vector () :
11.    waiting = false;
12.    new Thread(new Worker ()). start O ;
13.  }
14.
15.  public void put(RunnableTask r){
16.     tasks.add(r);
17.     if (waiting){
18.       synchronized (this){
19.            notifyAll();
20.       }
21.     }
22.  }
23.
24.  public RunnableTask take(){
25.     if (tasks.isEmpty()){
26.       synchronized (this)(
27.            waiting = true;
```

```
28.          try{
29.              wait ();
30.          } catch (InterruptedException ie) {
31.              waiting = false;
32.           }
33.       }
34.     }
35.     return (RunnableTask)tasks.remove(0);
36. }
37.
38.  private class Worker implements Runnable{
39.     public void run(){
40.       while (!shutdown){
41.           RunnableTask r = take();
42.          r.execute() ;
43.       }
44.     }
45. }
46.}
```

В данном примере используются два класса, реализующие интерфейс RunnableTask, — `AddressRetriever` (листинг 4.13) и `ContactRetriever` (листинг 4.14). Классы очень похожи друг на друга, оба используют технологию RMI для запроса бизнес-объекта у сервера. Как видно из их названий, каждый класс получает от сервера бизнес-объекты определенного типа, делая доступными клиентам объекты класса Address и Contact, соответственно.

Листинг 4.13. AddressRetriever.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. public class AddressRetriever implements RunnableTask{
4.   private Address address;
5.   private long addressID;
6.   private String url;
7.
8.   public AddressRetriever(long newAddressID, String newUrl){
9.      addressID = newAddressID;
10.     url = newUrl;
11. }
12.
13.  public void execute(){
14.     try{
15.       ServerDataStore dataStore = (ServerDataStore)Naming.lookup(url);
16.      address = dataStore.retrieveAddress(addressID) ;
17.     }
18.     catch (Exception exc){
19.     }
20. }
21.
22.  public Address getAddress(){ return address; }
23.  public boolean isAddressAvailable(){ return (address == null) ? false :
   true; }
24.}
```

Листинг 4.14. ContactRetriever.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. public class ContactRetriever implements RunnableTask{
4.   private Contact contact;
5.   private long contactID;
6.   private String url;
7.
8.   public ContactRetriever (long newContactID, String newUrl){
9.      contactID = newContactID;
10.     url = newUrl;
11.  }
12.
13.  public void execute(){
14.     try{
15.       ServerDataStore dataStore = (ServerDataStore)Naming.lookup(url);
16.       contact = dataStore.retrieveContact(contactID);
17.     }
18.     catch (Exception exc)(
19.     }
20.  }
21.
22. public Contact getContact(){ return contact; }
23. public boolean isContactAvailable() { return (contact == null) ? false :
    true; }
24.}
```

Callback

Свойства шаблона

Тип: обрабатывающий (поведенческий)
Уровень: архитектурный

Назначение

Обеспечение клиенту возможности регистрации на сервере для выполнения расширенных операций. Это позволяет серверу извещать клиента при завершении операции.

Представление

Работающее в сети **PIM-приложение** время от времени будет обращаться к серверу с запросами, требующими высоких накладных расходов. Например, трудно предсказать, сколько времени может понадобиться для получения всего проекта, хранящегося на сервере, поскольку такой проект может содержать тысячи объектов задач и продуктов.

Кроме того, в подобной ситуации сервер также может сталкиваться с ограничениями, связанными с необходимостью поддержания открытых сетевых соединений. Хотя в общем случае наличие открытых соединений, как правило, позволяет улучшить эффективность работы сервера, с другой стороны, постоянное поддержание

соединений для каждого клиента в открытом состоянии значительно ограничивает количество одновременно обрабатываемых клиентских запросов.

Вместо того чтобы требовать наличия постоянного соединения клиента и сервера, лучше позволить последнему контактировать с клиентом по завершении обработки клиентского запроса. Именно такой подход реализуется шаблоном Callback.

- Клиент отправляет запрос на получение проекта от сервера, предоставляя в запросе информацию, необходимую для организации обратного вызова (callback).
- Клиент разрывает соединение с сервером и позволяет последнему выполнить все операции, необходимые для получения проекта.
- Сервер, закончив выполнение полученной задачи, соединяется с клиентом и отправляет ему запрашиваемую информацию о проекте.

Достоинства такого решения заключаются как в экономии пропускной способности сети, так и в обеспечении более эффективного использования процессорного времени сервера. Кроме того, данный подход позволяет серверу использовать такие средства, как очередь запросов и приоритеты задач, что дает возможность эффективнее управлять загрузкой сервера и имеющимися в его распоряжении ресурсами.

Область применения

Шаблон Callback имеет смысл применять в клиент/серверных системах при наличии операций, требующих от клиента длительного ожидания, и при выполнении двух следующих условий **или только одного из них.**

- Требуется сохранить ресурсы сервера для поддержки активных соединений.
- Клиент может и должен продолжать работу до тех пор, пока не получит запрашиваемую информацию. Это обычно достигается путем применения на стороне клиента средств обеспечения простейшей многозадачности.

Описание

В некоторых распределенных системах серверу приходится при обслуживании запросов клиентов выполнять достаточно длительные операции. В таких системах синхронные коммуникации скорее всего не являются наилучшим решением. Если сервер поддерживает связь с клиентом во время обработки его запроса, он использует ресурсы, которые могли бы задействоваться для выполнения других задач, например, обмена данными с другим клиентом. Представим себе систему, пользователь которой хочет выполнить сложный запрос к сравнительно большой таблице базы данных. Пусть, например, таблица содержит информацию о потребителях и имеет более 10000 записей. В синхронной клиент/серверной системе клиентскому процессу приходится ждать и, возможно, довольно долго, пока сервер не завершит обработку запроса. Сервер, прежде чем сможет вернуть **данные** клиенту, должен выполнить запрос, а также ряд других операций по организации, форматированию и упаковке полученных данных.

Альтернатива состоит в создании системы, которая позволяет клиенту регистрироваться для получения извещения от сервера. Когда сервер завершает запрашиваемую операцию, он самостоятельно извещает об этом клиента. До тех пор, пока это не произошло, и клиент, и сервер могут свободно распоряжаться своими ресурсами, используя их для более продуктивных целей, чем поддержание определенного канала связи.

Данная возможность обеспечивается с помощью шаблона Callback, который позволяет организовывать асинхронные клиент/серверные коммуникации. Процесс обмена информацией при использовании этого шаблона состоит из трех простых этапов.

1. *Регистрация клиента.* Клиент обращается к серверу с запросом, предоставляя ему свою контактную информацию.

   Например, клиент подключается к серверу и обращается к нему с запросом. Обычно клиент запрашивает определенную информацию. Например, по поводу объемов продаж в 2002 финансовом году, или выполнения определенной операции, например, ввода имени пользователя в качестве участника лотереи. Поскольку клиент не ожидает от подобных запросов немедленного результата, он предоставляет серверу свою контактную информацию.

2. *Обработка запроса сервером.* Сервер обрабатывает запрос клиента и форматирует ответ если это нужно. В это время клиент может решать другие задачи, а сервер заниматься обменом информацией с другими клиентами.

3. *Обратный вызов клиента сервером.* Сервер, завершив обработку запроса клиента, отправляет последнему извещение. Это извещение обычно оформляется в соответствии с одной из двух следующих форм.

   - Информация, запрашиваемая клиентом. Такой подход обычно используется в тех случаях, когда клиенту нужны все полученные данные или когда данные имеют относительно небольшой объем.
   - Сообщение, извещающее клиента о том, что данные или их части стали доступными. Этот подход чаще применяется в тех случаях, когда требуется получить большой объем информации, поэтому клиент может получать данные по частям. Получение данных по частям может происходить по мере их появления или же в соответствии с решением, принятым клиентом о том, какие данные из полученных ему нужны, а какие — нет.

Использование шаблона Callback можно проиллюстрировать на примере того, как делает покупки отец с тремя сыновьями. Младший сын хочет приобрести новую модель трансформера, средний — портативный компьютер, а старший — недавно вышедшую из печати книгу по Java. При этом существуют следующие ограничения.

- Каждую из перечисленных вещей не так-то просто найти, особенно учитывая тот факт, что они продаются в разных магазинах.
- Сыновья могут спокойно ходить по магазинам не более 5 минут.
- Отец может искать одновременно одну или две вещи, если ему приходится держать в памяти параметры большего количества запланированных покупок, его производительность резко падает.

К счастью, отец может оставить сыновей в игровом зале, позволив им играть на автоматах или бороться друг с другом, или делать что-нибудь еще, что взбредет им в голову. Затем отец отправляется за первой покупкой, находит ее, покупает, приносит сыновьям, отправляется за следующей покупкой и т.д.

Шаблон Callback можно применять в самых разных приложениях.

- Популярные сегодня в Web программы-агенты могут использовать шаблон Callback для извещения клиента о завершении обработки запроса. Например, рассмотрим агент поиска работы, установленный на узле monster.com. Пользователь может ввести критерии поиска нужного ему места работы и отправить запрос. Спустя какое-то время сервер известит пользователя о том, что в базе данных появилась информация о вакансии, соответствующей заданным пользователем критериям.
- Приложения, которые интенсивно работают с базами данных, часто используют шаблон Callback для повышения количества обслуживаемых клиентов.
- Шаблон Callback можно применять в приложениях, у которых на стороне сервера применяются детализированные последовательности операций. Например, сервер, занимающийся обработкой заказов, часто после размещения заказа пользователем выполняет множество операций. Сервер проверяет наличие товара на складе, устанавливает корректность предоставленной информации об оплате и доставке, а также координирует работу с другими информационными системами — складскими, производственными, бухгалтерскими и почтовыми. Шаблон Callback позволяет серверу извещать клиента о состоянии заказа по мере выполнения каждого из перечисленных выше этапов. Поскольку на выполнение подобных операций могут уходить часы или даже дни, большинству потребителей также нравится применение шаблона Callback.

Реализация

Компонентная диаграмма шаблона Callback представлена на рис. 4.6.

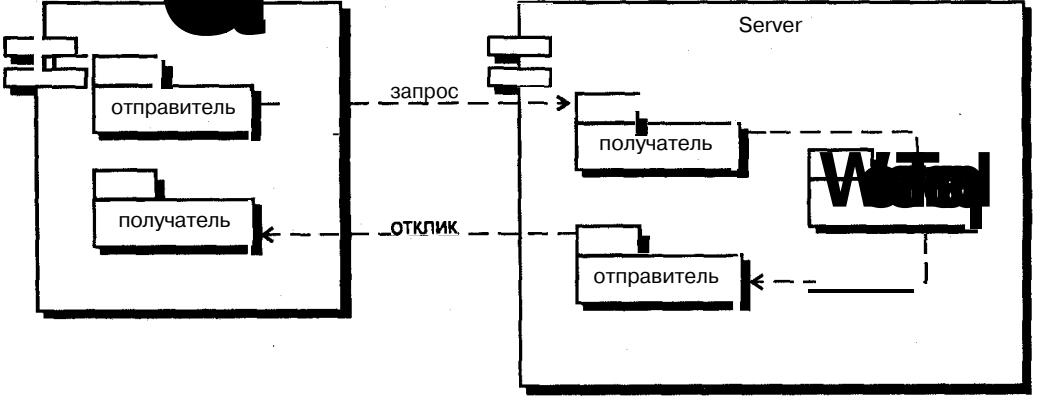


Рис. 4.6. Компонентная диаграмма шаблона Callback

Диаграмма последовательности событий для шаблона Callback представлена на рис. 4.7.

Шаблон Callback предъявляет определенные требования как к клиенту, так и к серверу.

- Клиент. Клиент должен предоставить серверу интерфейс обратного вызова, чтобы сервер мог установить контакт с клиентом по завершении обработки его запроса.
- Сервер. Помимо традиционного интерфейса вызова, которым пользуются клиенты, сервер должен иметь средства извещения клиентов по завершении обработки их запросов. Кроме того, сервер должен иметь возможность обрабатывать, а в случае необходимости и сохранять запросы клиентов.

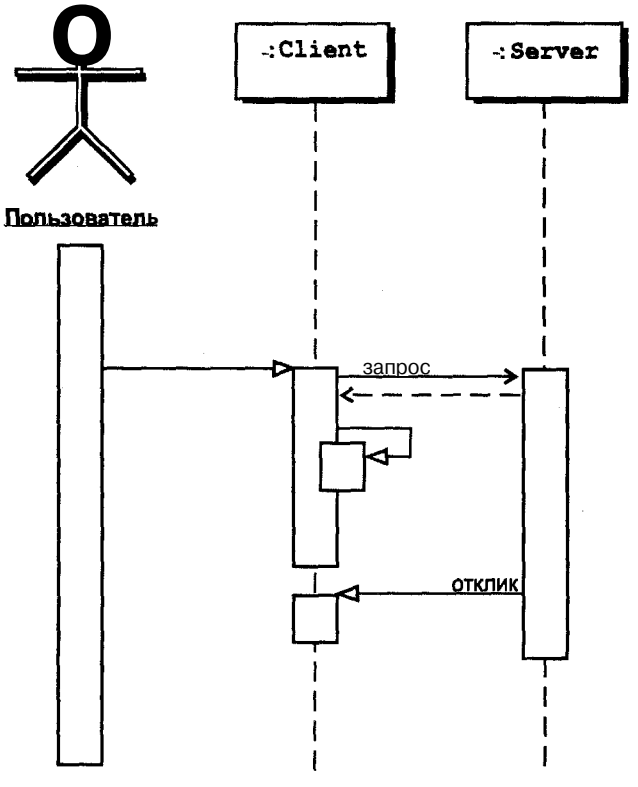


Рис. 4.7. Диаграмма последовательности событий для шаблона Callback

Достоинства и недостатки

Основным достоинством шаблона Callback является повышение эффективности работы системы, особенно производительности сервера. Улучшения проявляются в основном в двух областях.

- *Обработка на стороне сервера.* Серверу не приходится поддерживать работоспособность коммуникационных потоков, предназначенных для обслуживания ожидающих клиентов, поэтому он может перераспределить высвобождающиеся ресурсы на обработку клиентских запросов или на обслуживание других клиентов. Более того, обработка запросов может выполняться в тот момент, который сервер посчитает наиболее удобным, поскольку серверу не нужно выполнять запросы немедленно.
- *Серверные коммуникации.* Серверу не нужно поддерживать работоспособность открытых соединений, предназначенных для обслуживания ожидающих клиентов. Это означает, что сервер может обслужить большее количество клиентов, имея те же ограниченные ресурсы, например сокеты.

  Это является одним из самых важных побудительных мотивов для применения шаблона Callback. В тех случаях, когда нагрузка сервера слишком высока или непредсказуема (например, как в Web), данная возможность предоставляет в распоряжение разработчика целый ряд преимуществ. В самых крайних проявлениях это может выражаться в замене целой группы работающих параллельно серверов одной машиной, обрабатывающей все запросы пользователей с не меньшей эффективностью.

  Другое преимущество состоит в том, что клиентам не нужно ждать полного завершения обработки запроса сервером, а можно вернуться к решению других задач. Ожидая ответа от сервера, клиент может заниматься другими делами. Когда результат будет готов, он немедленно об этом узнает от сервера.

В зависимости от реализации шаблон Callback может обеспечивать постановку клиентских запросов в очередь, позволяя тем самым серверу лучше организовать и сбалансировать свою загрузку. Кроме того, шаблон, в принципе, позволяет серверу извещать клиентов об изменениях и после истечения срока действия клиента. Хорошим примером вышесказанного являются Web-агенты, поскольку они позволяют клиентам вводить запросы в одном сеансе работы, а получать извещения о результатах в другом сеансе.

Одной из типовых проблем при реализации шаблона Callback является то, что он требует от клиента постоянно ожидать обратного вызова сервера. Это часто приводит к усложнению программного кода клиентов, а также повышает загрузку клиентской системы. Другой недостаток шаблона следует из того, что шаблон Callback отделяет запросы от клиентов. Недостаток такого разделения проявляется в затруднении отмены или модификации запроса после его отправки серверу.

Варианты

Все варианты реализации шаблона Callback обычно сосредоточены вокруг выбора стратегии обработки запросов сервером и подходов к извещению клиентов. При реализации обработки запросов на стороне сервера чаще всего применяются два следующих подхода.

- *Прямая обработка.* В этом случае сервер создает обработчик потока для выполнения каждого клиентского запроса. Такой подход очень прост в реализации, но его иногда трудно масштабировать для большого количества клиентов, запрашивающих обслуживание.
- *Очередь запросов.* Сервер ведет очередь клиентских запросов и пул обработчиков потоков. Обработчики потоков (подробнее см. раздел "Worker Thread" на стр. 241) назначаются для выполнения обработки клиентских запросов по мере необходимости.

При реализации извещения клиентов могут применяться разные подходы, выбор которых зависит от требований, выдвигаемых приложением.

- *Активный обратный вызов.* Клиент использует серверный процесс, отслеживающий наличие входящих коммуникаций. Это позволяет клиенту непосредственно получать извещения от сервера.
- *Опрос клиентов.* При данном подходе клиентам приходится периодически проверять состояние обработки своего запроса. Когда запрос или его часть завершается, клиент запрашивает соответствующую информацию у сервера.
- *Явное уведомление.* Сервер может постоянно отправлять сообщения, пока не получит подтверждение пользователя. Этот подход иногда используется в тех случаях, когда обработка запроса сервером требует больше времени, чем длительность существования клиентского приложения. Хотя все вышесказанное не относится к TCP, поскольку сокет не может быть открыт при отсутствии клиента, такой подход может иметь место при использовании коммуникационных технологий, не гарантирующих доставку информации, например UDP.

Родственные шаблоны

К родственным можно отнести шаблон Worker Thread (стр. 241). Этот шаблон используется для обеспечения планирования обработки клиентских запросов. В соответствии с шаблономWorker Thread, запросы помещаются в очередь, а затем обработчик потоков выполняет их.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом RunPattern, приведен в разделе "Callback" на стр. 514 Приложения А.

В PIM-приложении имеется один элемент, размер которого может сильно изменяться, — это проект. Действительно, проект может состоят из нескольких задач, а может содержать сотни и даже тысячи отдельных этапов. В данном примере показано, как с помощью шаблона Callback можно организовать получение проекта, хранящегося на сервере.

Представленный в листинге 4.15 интерфейс CallbackServer содержит определение лишь одного метода сервера `getProject`. Обратите внимание на то, что этот метод требует, помимо идентификатора проекта, информацию для обратного вызова: имя клиентской машины и имя клиентского объекта RMI. Данный интерфейс реализуется классом `CallbackServerImpl` (листинг 4.16).

Листинг 4.15. CallbackServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface CallbackServer extends Remote{
4.   public void getProject(String projectID, String callbackMachine,
5.     String callbackObjectName) throws RemoteException;
6. }
```

Листинг 4.16. CallbackServerImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. public class CallbackServerImpl implements CallbackServer{
4.   private static final String CALLBACK_SERVER_SERVICE_NAME =
   "CallbackServer";
5.   public CallbackServerImpl(){
6.     try {
7.       UnicastRemoteObject.exportObject(this);
8.       Naming.rebind(CALLBACK_SERVER_SERVICE_NAME, this);
9.     )
10.    catch (Exception exc){
11.      System.err.println("Error using RMI to register the
   CallbackServerImpl" + exc);
12.    }
13. }
14.
15. public void getProject(String projectID, String callbackMachine,
16.    String callbackObjectName){
17.      new CallbackServerWorkThread(projectID, callbackMachine,
   callbackObjectName);
18. }
19.
20.}
```

В реализации метода `getProject` класс CallbackServerImpl делегирует задачу получения проекта рабочему объекту CallbackServerDelegate (листинг 4.17). Этот объект выполняется в отдельном потоке и отвечает за извлечение проекта и отправку его клиенту.

Листинг 4.17. CallbackServerDelegate.java

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. public class CallbackServerDelegate implements Runnable{
6.   private Thread processingThread;
7.   private String projectID;
8.   private String callbackMachine;
9.   private String callbackObjectName;
10.
11.  public CallbackServerDelegate(String newProjectID, String
   newCallbackMachine,
12.  String newCallbackObjectName){
13.     projectID = newProjectID;
14.     callbackMachine = newCallbackMachine;
15.     callbackObjectName = newCallbackObjectName;
16.     processingThread = new Thread(this);
17.     processingThread.start();
18. }
19.
20.  public void run(){
21.     Project result - getProject();
22.     sendProjectToClient(result);
23. }
24.
25.  private Project getProject(){
26.     return new Project(projectID, "Test project");
27. }
28.
29.  private void sendProjectToClient(Project project){
30.     tryf
31.       String url = "//" + callbackMachine + "/" + callbackObjectName;
32.       Object remoteClient = Naming.lookup(url);
33.       if (remoteClient instanceof CallbackClient){
34.           ((CallbackClient)remoteClient).receiveProject(project);
35.       )
36.     }
37.     catch (RemoteException exc)()
38.     catch (NotBoundException exc){}
39.     catch (MalformedURLException exc){}
40. }
41.}
```

В методе run класса CallbackServerDelegate осуществляется извлечение проекта с помощью вызова метода getProject с последующей отправкой его клиенту путем вызова метода sendProjectToClient. Последний метод осуществляет обратный вызов клиента— объект CallbackServerDelegate вызывает объект RMI типа CallbackClient, находящийся на клиентской машине. Интерфейс CallbackClient (листинг 4.18) также определяет один метод RMI receiveProject.

Листинг 4.18. CallbackClient.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface CallbackClient extends Remote{
4.   public void receiveProject(Project project) throws RemoteException;
5. }
```

Класс CallbackClientImpl (листинг 4.19), реализующий интерфейс Callback-Client, является одновременно и клиентом, и сервером. Его метод requestProject обращается к объекту класса CallbackServer и вызывает удаленный метод getProject. Класс также определяет удаленный метод receiveProject, который вызывается обработчиком потока сервера, когда проект готов для предоставления его клиенту. Класс CallbackClientImpl имеет булеву переменную projectAvailable, которая позволяет клиентской программе определить, готов ли проект для отображения.

Листинг 4.19. CallbackClientImpl.java

```
1. import java.net.InetAddress;
2. import java.net.MalformedURLException;
3. import java.net.UnknownHostException;
4. import java.rmi.Naming;
5. import java.rmi.server.UnicastRemoteObject;
6. import java.rmi.NotBoundException;
7. import java.rmi.RemoteException;
8. public class CallbackClientImpl implements CallbackClient{
9.  private static final String CALLBACK_CLIENT_SERVISE_NAME =
   "callbackClient";
10.  private static final String CALLBACK_SERVER_SERVICE_NAME =
   "callbackServer";
11.  private static final String CALLBACK_SERVER_MACHINE_NAME = "localhost";
12.
13.  private Project requestedProject;
14.  private boolean projectAvailable;
15.
16.  public CallbackClientImpl(){
17.    try {
18.      UnicastRemoteObject(this) ;
19.      Naming.rebind(CALLBACK_CLIENT_SERVISE_NAME, this);
20.    }
21.    catch (Exception exc) {
22.      System.err.println("Error using RMI to register the
   CallbackClientImpl" + exc);
23.    }
24. }
25.
26.  public void receiveProject(Project project){
27.    requestedProject = project;
28.    projectAvailable - true;
29. }
30.
31.  public void requestProject(String projectName){
32.    try{
33.      String url = "//" + CALLBACK_SERVER_MACHINE_NAME +"/" +
   CALLBACK_SERVER_SERVICE_NAME;
34.      Object remoteServer = Naming.lookup(url);
35.      if (remoteServer instanceof CallbackServer){
36.           ((CallbackServer)remoteServer).getProject(projectName,
37.                InetAddress.getLocalHost().getHostName(),
38.                CALLBACK_CLIENT_SERVICE_NAME);
39.      }
40.      projectAvailable = false;
41.    }
42.    catch (RemoteException exc){}
43.    catch (NotBoundException exc){}
44.    catch (MalformedURLException exc)(}
```

```
45.     catch (UnknownHostException exc){}
46.  }
47.
48.  public Project getProject(){ return requestedProject; }
49.  public boolean isProjectAvailable(){ return projectAvailable; }
50.}
```

Основные операции происходят в следующей последовательности. Когда клиент запрашивает проект, объект класса CallbackClientImpl вызывает метод getProject объекта класса `CallbackServerImpl`. Последний создает для извлечения проекта объект `CallbackServerWorkThread`. После того как поток CallbackServerWorkThread завершит свою работу, он вызывает метод клиента `receiveProject`, отправляя экземпляр класса Project запрашивавшему его объекту `CallbackClientImpl`.

Successive Update

Также известен как Client Pull / Server Push

Свойства шаблона

Тип: обрабатывающий (поведенческий)
Уровень: архитектурный

Назначение

Основное назначение заключается в обеспечении клиенту возможности постоянного получения обновлений от сервера. Такие обновления обычно отражают изменения данных сервера, появившиеся или обновленные ресурсы либо изменения в состоянии бизнес-модели.

Представление

Предположим, что требуется использовать PIM-приложение для координации работы нескольких пользователей. Например, можно позволить нескольким пользователям совместно использовать информацию о части проекта, над которой они работают. Если проект представляет собой совокупность отдельных объектов классов Task и Deliverable, разработчик может позволить некоторым пользователям (например, руководителю проекта) получать обновленные сведения о ходе выполнения одного или нескольких заданий.

Если в системе создается сервер, вполне естественно централизовать хранение информации о проекте. Иными словами, объекты классов Project, Task и Deliverable, по всей видимости, будут храниться на сервере. Однако как при этом обеспечить своевременное обновление информации для клиентов?

Можно выбрать одну из двух стратегий, каждая из которых представляет собой реализацию шаблона Successive Update. Можно возложить задачу регулярного опроса сервера на *клиента,* который будет постоянно запрашивать наличие обновлений своей задачи, или же обязать *сервер* рассылать обновления всех объектов класса `Task` клиентам.

В обоих случаях роль сервера в системе расширяется, в результате чего мы получаем решение, ориентированное на поддержку групповой обработки данных. Так или иначе, но клиенты периодически получают обновленную информацию, что гарантирует скоординированность выполняемой ими работы.

Областьприменения

Шаблон Successive Update рекомендуется использовать в клиент/серверных системах в следующих случаях.

- Ресурсы или данные сервера постоянно изменяются под воздействием нескольких клиентов или внешних обновлений.
- Необходимо организовать получение обновлений клиентом, не заставляя пользователя выполнять операцию обновления вручную.

Описание

Шаблон Successive Update используется в приложениях, которым требуется обеспечить постоянное обновление состояния клиента. Хотя, конечно же, можно обновлять данные вручную, такое решение нельзя назвать привлекательным и интересным для конечного пользователя, который должен будет постоянно помнить о необходимости обновить информацию. Достаточно представить, что скажет пользователь о системе, в которой ему нужно ежеминутно вручную отправлятьсерверу запрос на обновление своих данных. Единственное достоинство такой системы заключается в том, что спустя некоторое время кисти рук пользователя станут крепкими и мускулистыми, если, конечно, он вообще захочет работать с такой системой.

Для получения приемлемой альтернативны следует воспользоваться шаблоном Successive Update, который автоматизирует работу по обновлению, которую пользователю пришлось бы выполнять вручную. Клиент и сервер устанавливают стратегию автоматического обновления, освобождая тем самым пользователя от необходимости заниматься обновлением лично. Шаблон Successive Update можно реализовать поразному, но чаще всего его реализация привязывается к одной из технологий, представляющей двух участников процесса: запрос обновления клиентом (client pull) и рассылка обновлений сервером (server push).

Запрос обновления клиентом

При запросе обновления клиентом выполняется периодическое обновление информации под управлением программного кода на стороне клиента. Сервер работает в обычном режиме, предоставляя информацию, запрашиваемую его клиентами. С точки зрения сервера, запрос обновления клиентом ничем не отличается от обычных запросов информации, направляемых клиентом серверу.

Диаграмма последовательности событий для реализации шаблона Successive Update при запросе обновления клиентом представлена на рис. 4.8.

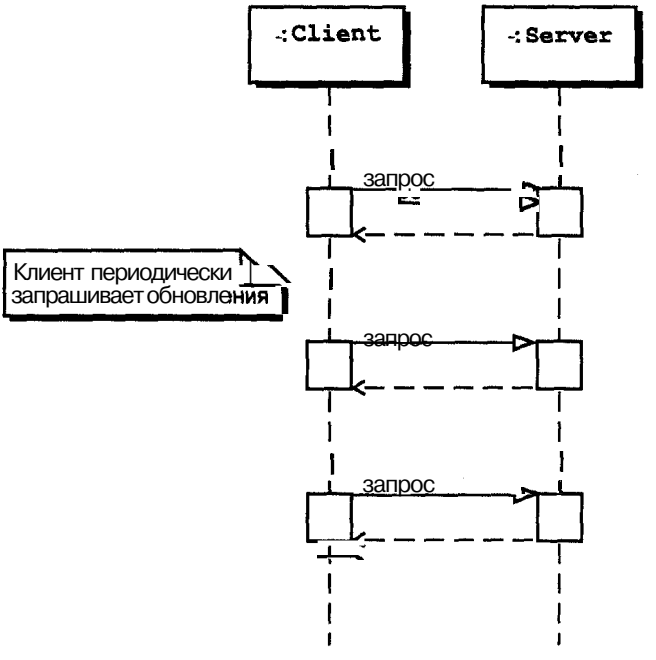


Рис. 4.8. Диаграмма последовательности событий шаблона Successive Update (запрос обновления клиентом)

Запрос обновления клиентом лучше всего применять в ситуациях, для которых справедливо хотя бы одно из следующих утверждений.

- При обновлении требуется передавать небольшой объем информации.
- Данные приложения постоянно меняются.
- Допускается некоторое запаздывание при обновлении информации.

Примеры применения запросов обновления клиентом в изобилии присутствуют в технологиях Web. Бегущая строка с котировками акций, бегущая строка с информацией о спортивных событиях, новостная бегущая строка и прочие элементы, использующие представление информации в виде бегущей строки, как правило, работают по технологии запроса обновления клиентом. Еще одной возможной областью применения этой технологии являются приложения, которые позволяют пользователям просматривать обновляемые ресурсы, например структуру удаленных каталогов.

Любой, кто когда-либо путешествовал с детьми на автомобиле, сталкивался с ярким примером технологии запроса обновления клиентом. Клиенты (в возрасте от 3 до 12 лет) периодически обращаются к серверу с запросами на обновление состояния. Запросы клиентов обычно имеют вид: "Мы уже приехали?" или "Можно остановиться? Я хочу пить". Интенсивность потока запросов обычно составляет 1-2 запроса в минуту.

Рассылка обновлений сервером

Решения, построенные на рассылке обновлений сервером, требуют, чтобы специальный процесс, выполняющийся на сервере, отправлял извещения об изменениях всем клиентам, заинтересованным в получении таких извещений. Это позволяет своевременно извещать клиентов при малейших изменениях в данных системы.

Диаграмма последовательности событий для реализации шаблона Successive Update при рассылке обновлений сервером представлена на рис. 4.9.

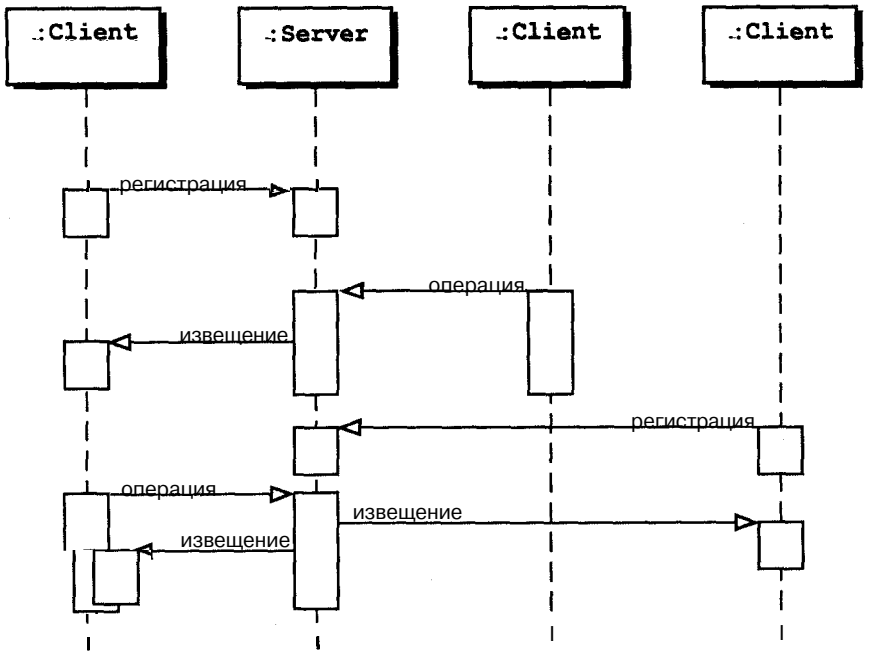


Рис. *4.9.* Диаграмма последовательности событий шаблона Successive Update (рассылка обновлений сервером)

Рассылку обновлений сервером лучше всего использовать в тех случаях, когда:

- клиентам требуется обновление в режиме реального времени (или в близком к нему);
- изменения данных происходят редко и в непредсказуемой последовательности.

Рассылка обновлений сервером часто используется в таких приложениях Web, как интерактивные игры или гостиные (chat room). В бизнес-приложениях эта технология также иногда применяется для обеспечения обновления совместно используемых ресурсов, например, документов, обрабатываемых средствами групповой работы (groupware).

Прекрасным примером реализации рассылки обновлений сервером являются списки рассылки (mailing list). Отправив запрос с просьбой включения себя в список рассылки компании, клиент (потребитель) получает от сервера (компании) серию сообщений, извещающих его о различных предлагаемых компанией продуктах. Клиент

может реагировать на эти извещения либо подписавшись на бумажную версию каталога товаров, либо найдя в Web тот узел, на котором он оформил подписку на список рассылки, чтобы немедленно отказаться от получения сообщений, постоянно забивающих его ящик.

Реализация

В зависимости от принятой стратегии, в реализации шаблона Successive Update выдвигаются разные требования к клиенту и серверу. При запросе обновления клиентом последний должен иметь возможность каким-то образом устанавливать периодичность, с какой он обращается к серверу. Часто этим процессом управляет специальный поток, выполняющийся на стороне клиента, который периодически опрашивает сервер и предоставляет клиенту полученные результаты. В такой реализации к серверу не выдвигается никаких дополнительных требований. Если же реализуется рассылка обновлений сервером, тогда необходимо, чтобы сервер каким-то образом вел список клиентов, заинтересованных в получении обновлений. При каждом событии, происходящем на сервере и о котором требуется оповестить клиентов, сервер должен рассылать клиентам извещения.

Достоинства и недостатки

Достоинства и недостатки шаблона Successive Update зависят от того, построена ли его реализация на технологии запроса обновления клиентом или на технологии рассылки обновлений сервером.

Каждая из стратегий может давать преимущество в снижении загрузки сервера в зависимости от частоты, с которой изменяются данные. Если данные меняются часто, меньшая загрузка сервера обеспечивается при запросе обновления клиентом. Если же скорость изменения данных невысока, более эффективное использование сервера обеспечивается при рассылке обновлений сервером.Кроме того, при рассылке обновлений сервером выполнение операций обновления оправдано в большей степени, что приводит к более высокой эффективности коммуникаций.

Недостатками запроса обновления клиентом является наличие некоторой задержки между реальным изменением данных на сервере и их обновления на стороне клиента, а также повышенный риск передачи избыточных данных. Клиент постоянно запрашивает обновление, даже если на сервере ничего не меняется, в результате чего клиенту постоянно пересылаются одни и те же данные. Оба недостатка основываются на том, что клиент не может точно определить, произошли ли на сервере какие-либо изменения. В случае рассылки обновлений сервером недостаток состоит как в увеличении загрузки сервера, так и в возможной отправке клиенту информации, которая ему не нужна. Сервер обычно рассылает клиентам информацию обо всех изменениях, поэтому требуется каким-то образом обеспечить доступ сервера к клиентам, что не всегда можно легко организовать из-за особенностей клиентского приложения или наличия брандмауэра.

Варианты

Наиболее распространенным вариантом реализации шаблона Successive Update является комбинированный подход к обновлению, объединяющий в себе как запрос обновления клиентом, так и рассылку обновлений сервером. В зависимости от требований, выдвигаемых к приложению, последнее может применять запрос обновления клиентом для рядовых, постоянно происходящих событий, а рассылку обновлений сервером — для извещений о критичных по времени реагирования событиях. Приложения, в которых применяются обе стратегии обновления, обычно определяют два вида извещений о критичных по времени событиях.

- Извещения о событиях, которые могут привести к возникновению ошибки в приложении, таких как удаление записи о потребителе.
- Извещения об обычных событиях, которые не приводят к возникновению ошибок или потере информации, например, о создании записи о новом потребителе.

Что касается рассылки обновлений сервером, специальный модуль сервера может передавать извещения строго определенному кругу клиентов, обеспечивая тем самым приемлемую длину списка клиентов, а также ограниченность объема передаваемой информации. При таком подходе по мере роста количества поддерживаемых клиентов нагрузка на сервер может быстро возрасти, хотя, конечно, и не с такой скоростью, как при опросе сервера клиентами. Для извещения множества клиентов часто предпочтительнее применять широковещательную рассылку информации. В этом случае сервер отправляет свои сообщения одному или нескольким серверам, которые транслируют их клиентам, зарегистрировавшимся в качестве заинтересованных получателей соответствующей информации. Данный подход широко используется в Web и технологиях смешанного вещания, что позволяет снизить нагрузку на исходный сервер.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Observer (стр. 111). Клиенты часто используют шаблон Observer для регистрации на сервере при реализации рассылки обновлений сервером.
- Callback (стр. 248). Реализация шаблона Successive Update может использовать шаблон Callback для организации рассылки обновлений сервером.
- Mediator (стр. 95). При реализации рассылки обновлений сервером последний часто работает в соответствии с шаблоном Mediator, отправляя обновления по мере их поступления заинтересованным клиентам.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`*, приведен в разделе "Successive Update" на стр. 520 Приложения* ***A****.*

В данном примере показана простая реализация технологии запроса обновления клиентом в PIM-приложении. Клиенты используют сервер для централизованного хранения информации о задачах, над которыми они работают. Каждый клиент следит за актуальностью своих данных, периодически запрашивая у сервера обновления.

В листинге 4.20 приведен пример программного кода, представленного классом `PullClient`, который получает для клиента сведения о задаче. Этот класс отвечает за поиск сервера RMI, через который он может постоянно получать информацию о задаче.

Листинг 4.20. PullClient.java

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. import java.util.Date;
6. public class PullClient{
7.   private static final String UPDATE_SERVER_SERVISE_NAME = "updateServer";
8.   private static final String UPDATE_SERVER_MACHINE_NAME = "localhost";
9.   private ClientPullServer updateServer;
10.  private ClientPullRequester requester;
11.  private Task updatedTask;
12.  private String clientName;
13.
14.  public PullClient(String newClientName){
15.     clientName = newClientName;
16.     try{
17.     • String url = "//" + UPDATA_SERVER_MACHINE_NAME + "/" +
   UPDATE_SERVER_SERVICE_NAME;
18.       updateServer = (ClientPullServer)Naming.lookup(url);
19.     }
20.     catch (RemoteException exc){}
21.     catch (NotBoundException exc){}
22.     catch (MalformedURLException exc({}
23.     catch (ClassCastException exc){}
24. }
25.
26.  public void requestTask(String taskID){
27.    requester = new ClientPullRequester(this, updateServer, taskID);
28. }
29.
30.  public void updateTask(Task task){
31.     requester.updateTask(task);
32. }
33.
34.  public Task getUpdatedTask(){
35.     return updatedTask;
36. }
37.
38.  public void setUpdatedTask(Task task) {
39.     updatedTask = task;
40.    System.out.println(clientName + ": received updated task: " + task);
41. }
42.
43.  public String toString(){
44.     return clientName;
45. }
46.}
```

Когда клиент желает получить обновление по текущей задаче, он вызывает метод `requestTask` экземпляра класса `PullClient`. Объект этого класса создает обработчик потока (см. раздел "Worker Thread" на стр. 241), класс которого представлен классом `ClientPullRequester` (листинг 4.21). Соответствующий объект потока выполняется на стороне клиента и регулярно обращается к серверу с запросами на обновление информации о задаче.

Листинг 4.21. ClientPullRequester.java

```
1. import java.rmi.RemoteException;
2. public class ClientPullRequester implements Runnable{
3.   private static final int DEFAULT_POLLING_INTERVAL = 10000;
4.   private Thread processingThread;
5.   private PullClient parent;
6.   private ClientPullServer updateServer;
7.   private String taskID;
8.   private boolean shutdown;
9.   private Task currentTask = new TaskImpl();
10.  private int pollingInterval = DEFAULT_POLLING_INTERVAL;
11.
12.  public ClientPullRequester(PullClient newParent, ClientPullServer
   newUpdateServer,
13.  String newTaskID) {
14.     parent = newParent;
15.     taskID = newTaskID;
16.     updateServer = newUpdateServer;
17.     processingThread = new Thread(this);
18.     processingThread.start();
19.  )
20.
21.  public void run(){
22.     while (!isShutdown()){
23.       try{
24.       currentTask = updateServer. getTask(taskID,
   currentTask.getLastEditDate() ) ;
25.            parent.setUpdateTask(currentTask);
26.       }
27.       catch (RemoteException exc){ }
28.       catch (UpdateException exc){
29.          System.out.println(" " + parent + ": " + exc.getMessage()) ;
30.       }
31.       try(
32.           Thread.sleep(pollingInterval) ;
33.       }
34.       catch (InterruptedException exc){}
35.     }
36.  )
37.
38.  public void updateTask(Task changedTask){
39.     try{
40.       updateServer.updateTask(taskID, changedTask);
41.     }
42.     catch (RemoteException exc){}
43.     catch (UpdateException exc){
44.       System.out.println(" " + parent + ": " + exc.getMessage());
45.     }
46.  }
47.
48.  public int getPollingInterval(){ return pollingInterval; }
49.  public boolean isShutdown (){ return shutdown; }
```

```
50.
51. public void setPollingInterval(int newPollingInterval){ pollingInterval =
    newPollingInterval; }
52. public void setShutdown(boolean isShutdown){ shutdown = isShutdown; }
53.}
```

Поведение сервера RMI определяется интерфейсом ClientPullServer (листинг 4.22), а управление им осуществляется с помощью класса ClientPullServerImpl (листинг 4.23). Два метода, getTask и updateTask, позволяют клиентам взаимодействовать с сервером.

Листинг 4.22. ClientPullServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. import java.util.Date;
4. public interface ClientPullServer extends Remote{
5.  public Task getTask(String taskID, Date lastUpdate) throws RemoteException,
    UpdateException;
6.  public void updateTask(String taskID, Task updateTask) throws
    RemoteException, UpdateException;
7. }
```

Листинг 4.23. ClientPullServerImpl.java

```
1. import java.util.Date;
2. import java.rmi.Naming;
3. import java.rmi.server.UnicastRemoteObject;
4. public class ClientPullServerImpl implements ClientPullServer{
5.  private static final String UPDATE_SERVER_SERVICE_NAME = "updateServer";
6.  public ClientPullServerImpl(){
7.     try{
8.       UnicastRemoteObject.exportObject(this);
9.       Naming.rebind(UPDATE_SERVER_SERVICE_NAME, this);
10.    }
11.    catch (exception exc) [
12.      System.err.println("Error using RMI to register the
    ClientPullServerImpl " + exc;
13.    }
14. }
15.
16. public Task getTask(String taskID, date lastUpdate) throws UpdateException{
17.    return UpdateServerDelegate.getTask(taskID, lastUpdate);
18. }
19.
20. public void updateTask(String taskID, Task updatedTask) throws
    UpdateException{
21.    UpdateServerDelegate.updateTask(taskID, updatedTask);
22. }
23.}
```

Поведение сервера, представленного классом ClientPullServerImpl, определяет класс UpdateServerDelegate (листинг 4.24). Именно он извлекает объекты класса Task, а также обеспечивает передачу обновленных копий этих объектов клиентам, выполняя сравнение последнего обновления с текущим обновлением по значению атрибута Date.

Листинг 4.24. UpdateServerDelegate.java

```
1. import java.util.Date;
2. import java.util.HashMap;
3. public class UpdateServerDelegate{
4.   private static HashMap tasks = new HashMap();
5.
6.   public static Task getTask(String taskID, Date lastUpdate) throws
   UpdateException{
7.       if (tasks.containsKey(taskID)){
8.        Task storedTask = (Task)tasks.get(taskID);
9.         if (storedTask.getLastEditDate().after(lastUpdate)){
10.             return storedTask;
11.        }
12.        else{
13.           throw new UpdateException("Task" + taskID + "does not need to be
   updated", UpdateException.TASK_UNCHANGED);
14.        }
15.     }
16.     else{
17.       return loadNewTask(taskID);
18.     }
19.  }
20.
21.  public static void updateTask(String taskID, Task task) throws
   UpdateException{
22.     if (tasks.containsKey(taskID)){
23.        if
    (task.getLastEditDate().equals(((Task)tasks.get(taskID)).getLastEditDate()))
    {
24.           ((TaskImpl)task).setLastEditDate(new Date());
25.            tasks.put(taskID, task);
26.        }
27.        else{
28.           throw new UpdateException("Task " + taskID + " data must be
   refreshed before editing", UpdateException.TASK_OUT_OF_DATE);
29.        )
30.     }
31. >
32.
33.  private static Task loadNewTask(String taskID){
34.     Task newTask = new TaskImpl(taskID, "", new Date(), null);
35.     tasks.put(taskID, newTask);
36.     return newTask;
37.  }
38.}
```

Router

Также известен как Request Router, Multiplexer

Свойства шаблона

Тип: обеспечивающий совместную работу
Уровень: архитектурный

Назначение

Отделение источников информации от ее получателей.

Представление

Если пользователь PIM-приложения является профессионалом и при этом приятным в общении человеком, легко предположить, что он будет интенсивно обмениваться информацией с другими пользователями, тем самым серьезно нагружая приложение. Это приведет к тому, что PIM-приложение будет отслеживать множество событий, причем многие из его компонентов будут тем или иным способом реагировать на одни и те же события или же извещаться о возникновении таких событий. Под событиями здесь понимаются не только "события" в том смысле, какой используется в графическом пользовательском интерфейсе, а вообще любые события. Например, начальнику пользователя понадобилось узнать, сможет ли он отменить все встречи, которые были запланированы в период с 2 до 3 часов дня, чтобы принять участие в совещании в его кабинете. Или же пользователь настроил приложение таким образом, чтобы оно известило его о падении курса акций определенной компании, так как он собирается купить 100 акций подешевле.

Несомненно, в таких сложных системах, как PIM-приложение, имеется множество источников информации. Количество получателей информации также может быть достаточно большим причем вполне вероятно, что существует множество объектов, которые получают информацию из одного источника. Для того чтобы упростить работу источников событий, а также освободить их от необходимости извещения всех потенциальных получателей информации, можно применить подход, состоящий в выделении функций по доставке событий в отдельную сущность. Этот подход реализуется шаблоном Router, поскольку он объединяет несколько источников событий и несколько получателей.

Не имея возможности идентифицировать отдельных клиентов, сервер будет ограничиваться только теми операциями, которые можно выполнить в виде одной операции. Это приведет к тому, что разработчику придется выбрать одно из двух возможных решений. Он может создать весьма объемную операцию (под "весьма объемной" мы понимаем такую операцию, которую можно увидеть из космоса), в которой будут содержаться все данные. Второе решение состоит в таком изменении модели приложения, которое позволило бы управлять операциями, не привязанными к конкретному состоянию, например, построить приложение вокруг системы проверки данных. Конечно же, это означает, что потребуется позаботиться о восстановлении системы после сбоев, а также о безопасности системы проверки. Если как следует поразмыслить, ни одно из двух решений нельзя назвать привлекательным, поскольку они оба требуют радикального изменения архитектуры приложения и способа взаимодействия пользователя с системой.

Область применения

Шаблон Router рекомендуется использовать в следующих случаях:

- когда имеется несколько источников информации;
- когда имеется несколько получателей информации, генерируемой одним или несколькими источниками.

Описание

В PIM-приложении информация хранится централизованно и совместно используется разными частями приложения, а также другими приложениями. При этом каждый из источников информации может заниматься ее распространением.

Таким образом, каждый получатель информации должен регистрироваться у ее источника, а каждый источник— вести список всех получателей, которых нужно уведомлять при возникновении изменений. Если разработчик захочет расширить свое приложение, воспользовавшись для отправки сообщений многопоточностью, ему придется реализовать этот механизм для каждого источника. Легко увидеть, что гораздо проще управлять распределением информации с помощью отдельного механизма, что и реализуется в соответствии с шаблоном Router.

Работу реализации шаблона Router можно сравнить с работой сетевого маршрутизатора. Маршрутизатор получает информацию из источника и определяет на основе того, откуда она поступила, куда ее нужно отправить. Для такого определения маршрутизатор должен иметь таблицу, в которой указывается, как соотносятся различные входные каналы (источники) с получателями информации. В этой таблице отражается последовательность управляющих воздействий по каждому маршруту от источника информации к ее получателю. Получатели могут добавляться и удаляться с помощью вызова соответствующих методов маршрутизатора, без обращения к источникам информации.

Учитывая объем информации, проходящей через маршрутизатор, его пропускная способность должна быть как можно больше, поскольку в противном случае возникнет заметное снижение производительности системы. Поэтому если какой-то маршрут перестает по каким-то причинам функционировать, маршрутизатор должен отключить соответствующего получателя. Реализация такого способа управления позволяет обеспечить качество обслуживания, необходимое для успешной доставки информации другим получателям.

Реализация

Диаграмма классов шаблона Router представлена на рис. 4.10.

При реализации шаблона Router обычно используются следующие классы.

- InputChannel. Может иметь любой тип. Используется классом Router для создания таблицы маршрутизации; другое название — источник.
- `OutputChanne` 1. Интерфейс, который определяет методы отправки сообщений.

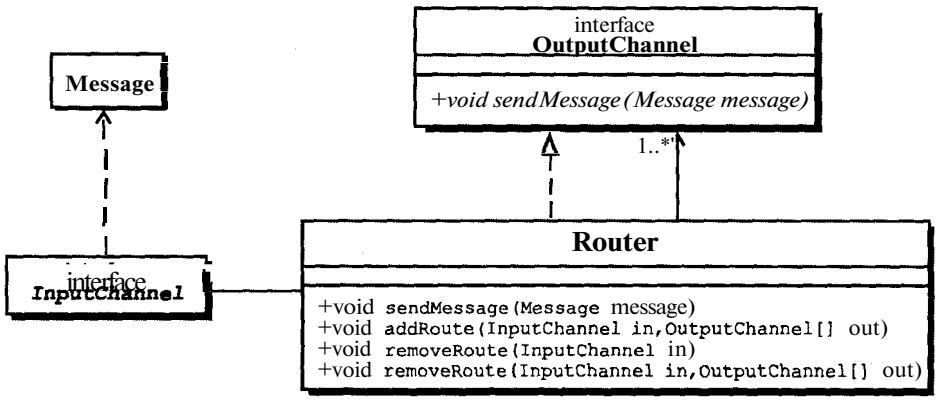


Рис. 4.10. Диаграмма классов шаблона Router

- Message. Класс, содержащий информацию, которая подлежит доставке. Для того чтобы позволить маршрутизатору сопоставить сообщение с определенным маршрутом, в сообщении содержится ссылка на источник.
- Router. Класс Router ведет таблицу маршрутизации, соединяющую входные каналы InputChannel с выходными OutputChannel, а также реализует интерфейс OutputChannel. Когда маршрутизатор получает сообщение, он пересылает его в определенные выходные каналы.

Достоинства и недостатки

Можно отметить следующие достоинства и недостатки шаблона Router.

1. Отделение источника от получателя. Источник информации не обязан знать, куда отправится сообщение, — ему достаточно лишь знать, где находится маршрутизатор. Все сведения о том, как выполняется маршрутизация между источником и получателем, известны маршрутизатору.
2. Наличие сбоев в одном канале вовсе не означает, что каким-то образом нарушается работа остальных каналов. Иными словами, один сбойный канал не может заблокировать маршрутизатор — он продолжит заниматься маршрутизацией сообщений между источниками и получателями.
3. Для организации входных и выходных каналов применяются различные стратегии. Маршрутизатор может выполнять каждый канал в отдельном потоке или же объединить в одном потоке все входные каналы.
4. Упрощается код клиентов, поскольку доставкой сообщений занимается маршрутизатор.
5. Повышается надежность, так как один отказавший канал не сможет заблокировать всю систему. Если канал не работает, он может игнорироваться либо могут приниматься другие меры, например, отключение проблемного канала. При этом маршрутизатор будет продолжать работу.

Варианты

Вариантом реализации шаблона Router является решение, состоящее в том, чтобы маршрутизатор вел таблицу соответствия между случайным образом заданным ключом и получателями информации. Таким образом, один и тот же источник может отправлять данные разным получателям, в зависимости от выполнения определенных условий. Например, если источник имеет два отдельных метода, у каждого из которых имеются собственные выходные каналы. При традиционной реализации шаблона Router на каждый входной канал приходится только один маршрут. Фокус заключается в том, чтобы создать ключ и зарегистрировать его в маршрутизаторе в качестве входного канала, таким образом подменив "реальный" входной канал.

Источник вызывает метод send маршрутизатора, передавая ему два параметра: ключ и само сообщение. Маршрутизатор читает ключ и отправляет сообщение в те выходные каналы, которые были выбраны в соответствии с ключом.

Родственные шаблоны

К родственным можно отнести следующие шаблоны.

- Mediator (стр. 95). Шаблон Router подобен шаблону Mediator. Различие состоит в том, что при реализации шаблона Mediator решение о том, куда нужно переслать сообщение, принимается на основе содержимого этого сообщения, поэтому реализация шаблона является специфичной для каждого приложения. Реализация шаблона Router принимает решение о пересылке сообщения, основываясь на его источнике.
- Observer (стр. 111). Шаблон Router можно сделать более гибким, используя шаблон Observer для регистрации получателей информации.
- Worker Thread (стр. 241). ШаблонWorker Thread часто применяется при реализации шаблона Router для повышения эффективности.

Пример

Примечание

Полный работающий код данного примера со вспомогательными классами, а также классом `RunPattern`, приведен в разделе "Router" на стр. 527 Приложения А.

Шаблон Router можно с успехом применять в нескольких местах рассматриваемого примера приложения. Практически в любой ситуации, когда имеется несколько получателей информации можно применить шаблон Router. По сути, маршрутизатор — это реализация структуры, подобной структуре объектов класса Listener.

В данном примере представлен исходный код класса `Message` (листинг 4.25). Этот класс представляет собой контейнер для источника (InputChannel, листинг 4.26) и собственно сообщения, которое здесь представлено переменной типа String.

Листинг 4.25. Message.java

```
1. import java.io.Serializable;
2. public class Message implements Serializable{
3.   private InputChannel source;
4.   private String message;
5.
6.   public Message(InputChannel source, String message);
7.     this.source = source;
8.     this.message = message;
9.   }
10.
11.  public InputChannel getSource(){ return source; }
12.  public String getMessage(){ return message; }
13.}
```

Листинг 4.26. InputChannel.java

```
1. import java.io.Serializable;
2. public interface InputChannel extends Serializable{}
```

Выходной канал OutputChannel (листинг 4.27) — это интерфейс, который определяет метод отправки сообщения получателю. Поскольку выходной канал может использоваться для коммуникаций между разными машинами, он определен как удаленный интерфейс.

Листинг 4.27. OutputChannel.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface OutputChannel extends Remote{
4.   public void sendMessage(Message message) throws RemoteException;
5. }
```

Класс Router (листинг 4.28) имеет хеш-таблицу, в которой хранятся связи, установленные между заданным входным каналом и различными выходными каналами. Когда экземпляр класса получает сообщение, он по этой таблице определяет получателя.

Экземпляр класса перебирает все элементы коллекции и отправляет сообщение каждому получателю. В данном примере класс Router для отправки сообщения каждому объекту класса OutputChannel создает обработчик потока (подробнее см. раздел "Worker Thread" на стр. 241). Для повышения эффективности подобных приложений часто используются пулы потоков.

Листинг 4.28. Router.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. import java.rmi.server.UnicastRemoteObject;
4. import java.util.HashMap;
5. public class Router implements OutputChannel{
6.   private static final String ROUTER_SERVICE_NAME = "router";
```

```
7.  private HashMap links = new HashMapO;
8.
9.  public Router () {
10.     try{
11.       UnicastRemoteObject.exportObject(this);
12.      Naming.rebind(ROUTER_SERVICENAME, this);
13.     )
14.    catch (Exception exc) {
15.      System.err.println("Error using RMI to register the Router " + exc);
16.     }
17. }
18.
19.  public synchronized void sendMessage(Message message) (
20.    Object key = message.getSource ();
21.    OutputChannel[] destinations = (OutputChannel[])links.get(key);
22.    new RouterWorkThread(message, destinations);
23. }
24.
25.  public void addRoute(InputChannel source, OutputChannel[] destinations) {
26.    links.put(source, destinations);
27. }
28.
29.  private class RouterWorkThread implements Runnable{
30.    private OutputChannel [] destinations;
31.    private Message message;
32.    private Thread runner;
33.
34.    private RouterWorkThread(Message newMessage, OutputChannel[]
   newDestinations){
35.      message = newMessage;
36.      destinations = newDestinations;
37.      runner = new Thread(this);
38.      runner.start();
39.    }
40.
41.    public void run() {
42.      for (int i - 0; i < destinations.length; i++){
43.           try{
44.               destinations[i].sendMessage(message) ;
45.           }
46.           catch(RemoteException exc){
47.              System.err.println("Unable to send message to " +
   destinations[i]);
48.           }
49.      }
50.    }
51. }
52.}
```

При использовании шаблона Router необходимо обращать внимание на размер рассылаемых сообщений. Как правило, сообщение должно быть как можно меньшим. Однако при работе с объектами языка Java легко упустить из вида одну особенность. Объект может ссылаться на другие объекты, которые в свою очередь могут ссылаться на другие объекты и т.д. Поэтому пересылка небольшого, на первый взгляд, объекта может на самом деле повлечь за собой пересылку огромного объема информации. Например, отправка объекта `java.awt.Button` — это не очень хорошая идея, поскольку произойдет сериализация и отправка всего графического интерфейса.

Это очень похоже на то, чем может обернуться покупка детской игрушки. На первый взгляд может показаться, что приобретение трансформера — это не очень боль-

шая трата, но со временем, оценив расходы на все аксессуары (дополнительный протектор, лазерный меч и т.п.), вы поневоле задумаетесь, не дешевле ли было купить ребенку красивый свитер.

Transaction

Свойства шаблона

Тип: обеспечивающий совместную работу
Уровень: архитектурный

Назначение

Группирование коллекций методов таким образом, чтобы они либо были все успешно выполнены, либо все завершились неудачно.

Представление

В объектно-ориентированном программировании часто приходится иметь дело с многочисленными экземплярами нескольких классов. Но иногда требуется рассматривать несколько объектов так, как если бы они были одним объектом, или, по крайней мере, хотя бы обеспечить неизменность состояния нескольких объектов до завершения определенной операции.

Одним из примеров такой ситуации является операция перевода денег с одного счета на другой. Особенностью этой операции является то, что должны быть выполнены обе ее составляющие: списание денег с одного счета и зачисление на другой счет. Если хотя бы одна из этих двух задач не может быть выполнена, вторая задача также не выполняется. При этом конечный результат обязательно должен быть сбалансированным: сумма, списанная с одного счета, должна быть целиком зачислена на второй счет и наоборот. Если с первого счета можно списать не всю сумму, а лишь ее часть (например, из-за превышения предела кредита по счету), ни по первому, ни по второму счету операции не выполняются.

Если рассчитывать на то, что обе операции всегда завершаются успешно, перевод денег можно осуществить очень легко и с весьма высокой степенью риска. Если перевод завершится неудачно, владелец одного из счетов может увидеть, как с его счета куда-то пропала определенная сумма денег, что с его точки зрения нельзя назвать счастливым случаем. Владельцу же другого счета, наоборот, может "повезти", если на его счет будет зачислена неизвестно откуда взявшаяся сумма (правда, банк это происшествие вряд ли назовет "везением"). Поэтому в подобных ситуациях необходимо принимать ряд мер, гарантирующих, что или все операции завершатся удачно, или ни одна из них не будет выполнена.

Именно в таких случаях и используется шаблон Transaction. Он обеспечивает выполнение или невыполнение всех связанных между собой операций. Что именно происходит в каждом конкретном случае, когда та или иная операция завершается успешно или неуспешно, зависит от реализации шаблона.

Область применения

Шаблон Transaction рекомендуется применять в следующих случаях.

- когда необходимо синхронизировать выполнение нескольких методов.
- когда нужно иметь возможность восстановления состояния системы после сбоя.

Описание

Существуют задачи, для выполнения которых должно обеспечиваться взаимодействие разных частей приложения. Когда хотя бы одна часть задачи остается невыполненной, все остальные части также не должны выполняться. Иными словами, все объединенные методы, участвующие в решении задачи, должны завершится либо успешно, либо неуспешно. Причем если какая-то часть задачи не может быть выполнена, все остальные части завершаются аварийно, а система должна быть возвращена в исходное состояние, в котором она пребывала до начала выполнения задачи.

Решение в подобной ситуации состоит в использовании шаблона Transaction. Каждый участник транзакции пытается выполнить свою часть задачи. Если один из участников не может по каким-то причинам этого сделать, об этом извещается диспетчер транзакции, который затем информирует остальных участников о необходимости вернуться к исходному состоянию. Диспетчером транзакции может быть любой объект, который прямо или опосредованно связан со всеми ее участниками.

После того как все участники проинформируют диспетчера транзакции об успешном завершении обновления, последний сообщает всем участникам о необходимости подтвердить внесенные ими изменения. *Подтверждение* (commit) означает, что сохраненное участниками временное состояние системы должно стать постоянным ее состоянием.

Если какой-то участник не может выполнить свою часть задачи, диспетчер отменяет транзакцию и вызывает метод отмены, имеющийся у каждого участника. Участники транзакции возвращаются к своему исходному состоянию, в котором они находились до ее начала. Эта операция, которую также называют *откатом* (roll-back), может выполняться также в тех случаях, когда диспетчер транзакций решает отменить ее даже тогда, когда все ее участники завершили выполнение своих частей задачи успешно. Результат получается таким же, как и при сбое одного из участников, — выполняется откат вызванной транзакции.

Для обозначения транзакции может выпользоваться специальный идентификатор, роль которого может исполнять случайно выбранное длинное целое число или какой-то объект. Преимущество использования объекта, а не длинного целого в качестве идентификатора состоит в том, что объект может не только содержать информацию, но и обеспечивать определенное поведение. С другой стороны, длинное целое число, несмотря на то, что оно не может обладать никаким поведением, более выгодно с точки зрения объема памяти, необходимой для его хранения. К тому же, пересылка по сети длинного целого числа требует значительно меньших накладных расходов, чем пересылка объекта.

Обычно транзакция выполняется в следующей последовательности.

- Создается идентификатор транзакции (длинное целое число или специальный объект).
- Подключаются все участники транзакции, причем если хотя бы одного из участников не удается подключить, транзакция тут же завершается неудачно.
- Осуществляется попытка выполнения транзакции путем вызова всех необходимых прикладных методов или метода отмены в том случае, если любой из участников не может успешно выполнить свою часть задачи.
- Если все участники выполнили свою часть работы успешно, вызывается метод подтверждения, имеющийся у всех участников транзакции.

Реализация

Диаграмма классов шаблона Transaction представлена на рис. 4.11.

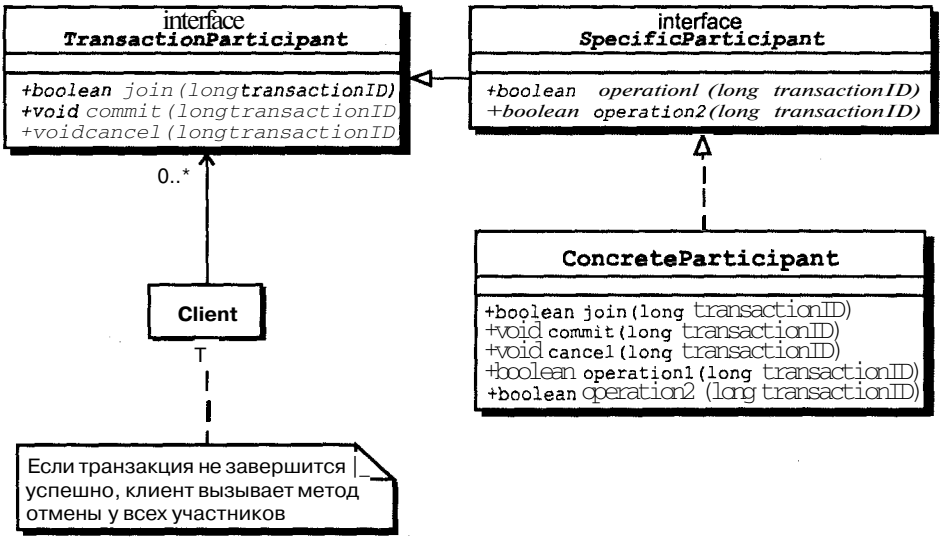


Рис. 4.11. Диаграмма классов шаблона Transaction

При реализации шаблона Transaction обычно используются следующие классы.

- `TransactionParticipant`. Интерфейс, который определяет методы, обеспечивающие управление любым участником транзакции.
- `SpecificParticipant`. Интерфейс, представляющий собой расширение общего интерфейса, который содержит прикладные методы. Все методы, участвующие в транзакции имеют один обязательный параметр, представляющий ее идентификатор. Кроме того, методы, участвующие в транзакции, могут генерировать исключительную ситуацию, сигнализируя о сбое.

- ConcreteParticipant. Реализация интерфейса SpecificParticipant, определяющая, что должно произойти, если диспетчер транзакции (в данном случае представлен классом Client), решит выполнить откат или подтверждение. С этой целью класс хранит ссылку на оригинальное состояние, чтобы иметь возможность восстановить его в случае вызова метода отмены.
- Client. Диспетчер транзакции. Клиент начинает транзакцию, вызывая метод подключения к транзакции у всех ее участников, а также обеспечивает ее откат или успешное завершение, вызывая методы отмены или подтверждения, соответственно.

Достоинства и недостатки

Очевидное достоинство данного шаблона состоит в том, что несколько методов можно объединить для того, чтобы они выполнялись, как одна атомарная операция. В результате такого подхода приложение всегда будет находиться в стабильном состоянии, поскольку новое состояние будет устанавливаться только тогда, когда все участники транзакции завершат свои операции удачно.

Недостаток шаблона Transaction состоит в снижении производительности. Если объект уже участвует в транзакции, а в это время вызывается его метод подключения к другой транзакции, объекту придется принимать какое-то решение о том, как поступить в данной ситуации. В большинстве случаев объекты генерируют исключительную ситуацию, поступающую диспетчеру транзакции, который вызвал метод подключения. Диспетчер транзакции может либо предпринять откат второй транзакции, либо подождать, пока занятый участник освободится.

Варианты

1. *Двухэтапное подтверждение.* Диспетчер транзакции, прежде чем вызывать методы подтверждения участников транзакции, желает убедиться, что все они могут выполнить этот метод. Поэтому прежде чем вызывать метод commit, диспетчер транзакции проводит опрос (voting round), в ходе которого каждый из участников сообщает диспетчеру, может ли он выполнить операцию подтверждения (первый этап). Если какой-то из участников не сможет этого сделать, транзакция отменяется с вызовом метода cancel всех участников (выполняется откат транзакции). Если все участники сообщили о своей готовности выполнить подтверждение, вызываются их методы commit (второй этап).

   Возможен вариант, при котором используется этап, предваряющий подтверждение. В этом случае выполняется проверка того, могут ли все участники выполнить подтверждение, которая является сигналом того, что следующий сигнал будет означать либо отмену, либо подтверждение.

2. *Оптимистические и консервативные транзакции.* При реализации механизма выполнения транзакций возможно применение одного из двух подходов: оптимистического или консервативного. Выбранный подход влияет практически на все звенья реализации, хотя необходимо отметить, что в чистом виде каждый из подходов встречается довольно редко.

Основное различие между этими подходами состоит в том, что при оптимистическом подходе участники всегда могут присоединиться к транзакции, но далеко не всегда могут выполнить подтверждение, тогда как при пессимистическом подходе участник не всегда может присоединиться к транзакции, но если ему это удалось, он всегда сможет выполнить подтверждение.

Кроме того, оба подхода различаются и в отношении к процессу присоединения к транзакции. При консервативном подходе клиент сначала должен вызвать метод `join` участника, чтобы подключить его к определенной транзакции, а затем может вызывать любые прикладные методы. При оптимистичном подходе участник сам вызывает метод `join`, если клиент, обращающийся к его прикладным методам, еще этого не сделал.

Родственные шаблоны

Отсутствуют.

Пример

Полный работающий код данного примера *со* ***вспомогательными классами,*** *а* ***также классом*** *RunPattern, приведен в разделе "Transaction" на стр. 534 Приложения А.*

PIM-приложение сохраняет информацию о запланированных событиях, основываясь на дате их проведения. Естественно, поскольку пользователи ведут активный образ жизни, в их планах постоянно происходят изменения. Поэтому еженедельник пользователя постоянно обновляется для отражения новых планов или внесения изменений в уже имеющиеся.

Когда нескольким пользователям требуется согласовать дату проведения какого-то мероприятия, было бы неплохо, чтобы такое согласование выполнялось автоматически их еженедельниками, которые бы выбрали дату, устраивающую всех пользователей. Именно такая задача и решается в данном примере — с помощью шаблона Transaction создается механизм согласования даты проведения мероприятия, встраиваемый в подсистему PIM-приложения, которая выполняет функции еженедельника.

Базовым интерфейсом, обеспечивающим поддержку транзакций, является `AppointmentTransactionParticipant` (листинг 4.29). Он содержит определение трех методов, обеспечивающих управление транзакцией (`join`, commit и `cancel`), а также одного прикладного метода changeDate. Этот интерфейс расширяет интерфейс Remote, поскольку он используется для организации обмена информацией между участниками транзакции, которые в общем случае могут выполняться в разных экземплярах виртуальной машины Java.

Листинг 4.29. AppointmentTransactionParticipant.java

```
1. import java.util.Date;
2. import java.rmi.Remote;
3. import java.rmi.RemoteException;
4. public interface AppointmentTransactionParticipant extends Remote{
5.  public boolean join(long transactionID) throws RemoteException;
```

```
6.  public void commit(long transactionID) throws TransactionException,
   RemoteException;
7.  public void cancel(long transactionID) throws RemoteException;
8.  public boolean changeDate(long transactionID, Appointment appointment,
9.     Date newStartDate) throws TransactionException, RemoteException;
10.}
```

Класс AppointmentBook (листинг 4.30), реализующий интерфейс AppointmentTransactionParticipant, представляет в приложении еженедельник пользователя. Помимо обеспечения изменения даты запланированного события, класс AppointmentBook может инициировать изменение самого события. Метод changeAppointment этого класса получает в качестве параметров: идентификатор транзакции; объект Appointment; массив ссылок на другие экземпляры класса AppointmentBook, участвующие в транзакции; массив возможных дат проведения запланированного мероприятия. Метод changeAppointment позволяет одному из объектов класса AppointmentBook обмениваться информацией с остальными объектами по технологии RMI, вызывая метод changeDate каждого из участников транзакции до тех пор, пока не будет согласована дата проведения запланированного мероприятия.

Листинг 4.30. AppointmentBook.java

```
1. import java.util.ArrayList;
2. import java.util.HashMap;
3. import java.util.Date;
4. import java.rmi.Naming;
5. import java.rmi.server.UnicastRemoteObject;
6. import java.rmi.RemoteException;
7. public class AppointmentBook implements AppointmentTransactionParticipant{
8.  private static final String TRANSACTION_SERVICE_PREFIX =
   "transactionParticipant";
9.  private static final String TRANSACTION_HOSTNAME = "localhost";
10. private static int index = 1;
11. private String serviceName = TRANSACTION_SERVICE_PREFIX + index++;
12. private HashMap appointments = new HashMap();
13. private long currentTransaction;
14. private Appointment currentAppointment;
15. private Date updateStartDate;
16.
17. public AppointmentBook(){
18.    try {
19.      UnicastRemoteObject.exportObject(this);
20.     Naming.rebind(serviceName, this);
21.    }
22.    catch (Exception exc){
23.      System.err.println("Error using RMI to register the AppointmentBook "
   +exc);
24.    }
25. }
26.
27. public String getUrl(){
28.   return "//" + TRANSACTION_HOSTNAME + "/" + serviceName;
29. }
30.
31. public void addAppointment(Appointment appointment))
32.    if (!appointments.containsValue(appointment)){
33.      if (!appointments.containsKey(appointment.getStartDate())){
34.          appointments.put(appointment.getStartDate(), appointment);
35.      }
```

```
36.    }
37.  }
38.  public void removeAppointment(Appointment appointment){
39.    if (appointments.containsValue(appointment)){
40.     appointments.remove(appointment.getStartDate() ) ;
41.    }
42.  }
43.
44.  public boolean join(long transactionID){
45.    if (currentTransaction != 0){
46.      return false;
47.    } else {
48.      currentTransaction = transactionID;
49.      return true;
50.    }
51.  }
52.  public void commit(long transactionID) throws TransactionException{
53.    if (currentTransaction != transactionID){
54.      throw new TransactionException("Invalid TransactionID");
55.    } else {
56.      removeAppointment(currentAppointment);
57.      currentAppointment.setStartDate(updateStartDate);
58.      appointments.put(updateStartDate, currentAppointment);
59.    }
60.  )
61.  public void cancel(long transactionID){
62.    if (currentTransaction == transactionID){
63.      currentTransaction = 0;
64.     appointments.remove(updateStartDate) ;
65.    )
66.  )
67.  public boolean changeDate(long transactionID, Appointment appointment,
68.    Date newStartDate) throws TransactionException{
69.      if ((appointments.containsValue(appointment)) &&
   (!appointments.containsKey(newStartDate))!
70.          appointments.put(newStartDate, null);
71.          updateStartDate = newStartDate;
72.          currentAppointment = appointment;
73.          return true;
74.     }
75.    return false;
76.  }
77.
78.  public boolean changeAppointment(Appointment appointment, Date[]
   possibleDates,
79.  AppointmentTransactionParticipant[] participants, long transactionID)!
80.    try{
81.      for (int i = 0; i< participants.length; i++)(
82.           if (!participants[i] .join(transactionID)){
83.                return false;
84.           }
85.      }
86.      for (int i = 0; i < possibleDates.length; i++){
87.           if (isDateAvailable(transactionID, appointment,
   possibleDates[i],participants)){
88.                try{
89.                     commitAll(transactionID, participants);
90.                     return true;
91.                )
92.               catch(TransactionException exc)( }
93.          }
94.     }
95.   }
```

```
96.     catch (RemoteException exc){ }
97.     try{
98.       cancelAll(transactionID, participants);
99.     }
100.    catch (RemoteException exc){)
101.    return false;
102. }
103.
104. private boolean isDateAvailable(long transactionID, Appointment
   appointment,
105. Date date, AppointmentTransactionParticipant[] participants){
106.    try{
107.     for (int i = 0; i < participants.length; i++){
108.          try)
109.              if (!participants[i] .changeDate(transactionID, appointment,
   date)){
110.                    return false;
111.               )
112.          }
113.          catch (TransactionException exc){
114.               return false;
115.          }
116.     }
117.    }
118.    catch (RemoteException exc){
119.      return false;
120.    )
121.    return true;
122. }
123. private void commitAll(long transactionID,
   AppointmentTransactionParticipant[] participants)
124.    throws TransactionException, RemoteException{
125.    for (int i = 0; i < participants.length; i++)(
126.     participants[i] .commit(transactionID) ;
127.    }
128. }
129. private void cancelAll(long transactionID,
   AppointmentTransactionParticipant[] participants)
130.    throws RemoteException{
131.    for (int i = 0; i < participants.length; i++){
132.     participants[i].cancel(transactionID) ;
133.    }
134. }
135. public String toString(){
136.    return serviceName + " " + appointments.values().toString();
137. }
138. }
```

Часть II

ШАБЛОНЫ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA

ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA

Глава 5

В первой части данной книги обсуждались шаблоны, образующие типовой набор используемых разработчиками решений, не привязанных к какому-то конкретному языку программирования. Обсуждение каждого шаблона сопровождалось описанием его характеристик, достоинств и недостатков, а также иллюстрировалось примерами его использования, подкрепленными исходным текстом небольших программ, призванных продемонстрировать применение шаблона на практике.

Все обсуждавшиеся в первой части базовые шаблоны не зависят от платформы и, как уже отмечалось, не привязаны ни к каким конкретным языкам программирования. Конечно, свойства языка Java делают его наиболее подходящим кандидатом на реализацию некоторых из рассмотренных шаблонов, но, тем не менее, они могут быть реализованы и на любом другом языке программирования — достаточно лишь, чтобы такой язык обладал средствами поддержки таких свойств объектно-ориентированного программирования, как наследование, инкапсуляция, полиморфизм и абстрактные классы.

Теперь пришло время углубиться в предмет и рассмотреть, как используются шаблоны в прикладных программных интерфейсах (API — Application Programming Interface) языка Java. Поскольку и языка Java, и движение сторонников шаблонов проектирования возникли и развивались практически одновременно, неудивительно, что разработчики языка Java интенсивно использовали шаблоны проектирования при создании API этого языка. Целью изучения материала, изложенного в данной части, является получение ответов на вопросы, которые позволят глубже понять принципы работы Java. К таким вопросам можно, например, отнести следующие.

- Каким образом в языке программирования Java обеспечивается использование шаблонов?
- Как применение шаблонов в этом языке позволяет повысить эффективность API?

Многие API языка Java представляют собой прекрасное наглядное пособие по использованию шаблонов. Подобно рассматривавшимся в первой части книги примерам, которые демонстрировали различные аспекты создания приложения для управления личной информацией (PIM — personal information manager), эти исполь-

зуемые на практике реализации шаблонов дают весьма полезное представление о том, как наилучшим образом применять шаблоны для решения различного рода проблем.

Здесь нужно сделать отступление и оговориться, что понимается под термином "API". Дело в том, что за последние годы толкование этого термина стало довольно расплывчатым. Сегодня под API может пониматься как один класс, так и группа классов, пакет или набор взаимосвязанных пакетов. Важнейшей характеристикой, позволяющей квалифицировать тот или иной программный код как API, является предоставление этим кодом набора взаимосвязанной функциональности, образующей некоторую программную базовую среду.

Многие из рассмотренных в последующих главах API разрабатывались в виде набора взаимосвязанных классов. Для того чтобы облегчить восприятие API как единого целого, имеет смысл потратить некоторое время на обсуждение базовой структуры этого программного интерфейса. Это позволяет лучше увидеть, как обеспечивается реализация шаблонов в API и как их применение помогает тому или иному API решать стоящие перед ним задачи. Таким образом, оставшиеся главы книги — это отчасти изучение шаблонов и отчасти оценка архитектурных решений. Выбранный нами подход к изложению материала данной части обладает несколькими практическими преимуществами.

1. Он может помочь вам в понимании того, как шаблоны используются на практике в API. Изучение API языка Java позволяет продемонстрировать, как можно применить шаблоны для достижения тех или иных практических целей.
2. Он показывает, как можно использовать шаблоны вместе с API. Изучение набора API позволит вам увидеть, как можно повысить эффективность приложения за счет реализации шаблонов, взаимодействующих с API или программной базовой средой.

В последующих главах мы рассмотрим несколько API языка Java, их устройство и применение. Это должно помочь вам выйти на новый уровень понимания их практической пользы, а также помочь разобраться, почему они устроены так, а не иначе. Итак, поставив перед собой перечисленные выше цели, давайте пойдем дальше.

Примечание

Каждая из последующих глав, посвященных рассмотрению прикладных программных интерфейсов, имеет определенную структуру, облегчающую изучение материала. Первым разделом каждой главы является раздел "Пакеты". В этом разделе перечислены пакеты, в которых содержатся классы и интерфейсы, образующие API. Следующий раздел, "Обзор", представляет собой краткое описание того или иного API. Этот раздел не преследует цели представить детальную картину внутреннего устройства API. Он лишь помогает тем, кто уже знаком с соответствующим API, вспомнить его основные свойства, а остальным — акцентировать внимание на ключевых моментах данного API, которые послужат отправными точками в процессе дальнейшего изучения. Последний раздел, "Использование шаблонов", содержит описание шаблонов, примененных в API, а также особенности их использования.

ОСНОВНЫЕ API ЯЗЫКА JAVA

Глава 6

Обработка событий

Пакеты

`java.awt.event`, `javax.swing.event`, `java.util` (некоторые классы)
Используется в J2SE (модель делегирования, впервые примененная в JDK 1.1)

Обзор

Обработка событий (event handling) — это механизм, позволяющий двум или более объектам системы обмениваться информацией об изменении состояния. В системах, управляемых событиями, один объект всегда выступает в качестве генератора событий, создавая объекты событий, предназначенные для обозначения определенных изменений в состоянии этого объекта. Затем генератор событий передает событие одному или нескольким зарегистрированным получателям, вызывая предназначенный для этого метод каждого объекта-получателя.

Средства поддержки обработки событий были реализованы в языке Java изначально. Они входят в качестве составной части в AWT, а впоследствии, начиная с выхода J2SE версии 1.2, они были реализованы и в Swing. Поскольку обработка событий играет важную роль во многих API Java, включая AWT, Swing и JavaBeans, средства ее поддержки были реализованы в пакете `java.util`.

Начиная с JDK 1.1 обработка событий строится в соответствии с так называемой моделью делегирования (delegation model). Она проектировалась как простая и гибкая модель, призванная обеспечить более высокий уровень надежности приложений. С помощью этой модели можно отделить прикладную часть программного кода от кода, обеспечивающего работу графического пользовательского интерфейса, что позволяет изменять обе части вне зависимости одной от другой. Введение специализированных типов событий и *обработчиков событий* (listener) позволило реализовать поддержку проверки типов во время выполнения приложения.

Обработка событий выполняется следующим образом. Для каждого типа события создается соответствующий класс вида xxxxxxEvent. Источник события (класс, который может генерировать экземпляры данного класса при возникновении заданного события), такой, например, как класс Component, хранит список зарегистрировавшихся у него обработчиков событий. При возникновении события источник создает экземпляр соответствующего класса и передает его тому обработчику, который должен обработать это событие. Для того чтобы найти нужный обработчик, источник вызывает у всех известных ему обработчиков специальный метод, передавая этому методу экземпляр класса события в качестве параметра.

Все классы событий представляют собой расширение класса `java.util.EventObject`, в котором определено поле, предназначенное для хранения ссылки на источник события. Некоторые события добавляют в класс дополнительную функциональность. Например, класс ActionEvent содержит поле `ActionCommand`, значение которого может устанавливаться источником события.

Обработчики событий — это интерфейсы, в которых содержатся определения методов, вызываемых источниками событий при возникновении определенных обстоятельств (щелчок на кнопке, перемещение мыши и т.п.). Все подобные интерфейсы расширяют интерфейс `java.util.EventListener`.

Источники событий не должны реализовывать какой-то определенный интерфейс или расширять те или иные классы. Однако все источники должны соответствовать одному требованию — им необходимо сохранять список обработчиков событий, зарегистрировавшихся для получения событий от данного источника. Обработчики событий регистрируются у источника с помощью вызова метода вида `addxxxxxxListener (xxxxxxListener)`. Для удаления обработчика из списка необходимо вызвать метод вида removexxxxxxListener (xxxxxxListener). При вызове конкретных методов обозначение xxxxxx заменяется именем конкретного события.

В общем случае источники событий могут быть как однонаправленными (т.е. иметь только один обработчик событий), так и многонаправленными (работать с несколькими обработчиками). В AWT и Swing все источники событий являются многонаправленными (multicast). Для того чтобы упростить задачу для разработчиков, был создан специальный класс многонаправленной диспетчеризации событий, определенный в пакете `java.awt.AWTEventMulticaster`. Класс AWTEventMulticaster обеспечивает эффективную и выполняющуюся в отдельном потоке многонаправленную диспетчеризацию событий AWT, определенных в пакете `java.awt.event` (подробнее см. в [JLS]).

Класс AWTEventMulticaster реализует все интерфейсы обработчиков событий AWT, поэтому его можно использовать для обработки любого события AWT. Конструктор класса AWTEventMulticaster позволяет проинициализировать два обработчика событий. Когда нужно добавить новый обработчик, создается новый экземпляр класса AWTEventMulticaster, конструктору которого в качестве параметров передаются ссылки на имеющийся экземпляр класса AWTEventMulticaster, а также на новый обработчик. Таким образом обеспечивается создание взаимосвязи всех экземпляров класса AWTEventMulticaster. Для того чтобы передать событие нужному обработчику, каждый экземпляр вызывает специальный метод обработчика у каждого из двоих дочерних объектов: один из них является обработчиком некоторого события, а другой— следующим экземпляром класса AWTEventMulticaster. Благодаря этому механизму обеспечивается многонаправленность обработки событий.

В Swing для этих целей используется другой класс, отличный от AWTEventMulticaster. Этот класс, включенный в пакет javax.swing.event.EventListenerList, может даже сохранять ссылки на обработчики событий неизвестного типа. Когда нужно, чтобы обеспечивалась работа одного источника событий с несколькими обработчиками событий разного типа, список этих обработчиков предоставляет класс EventListenerList.

Обработка событий предусмотрена и в пакетах, не входящих в состав API графического пользовательского интерфейса. В частности, таким пакетом является org.xml.sax, который впервые появился в Java версии 1.4. Технология SAX используется для анализа документов XML, а обработка событий позволяет организовать извещение различных обработчиков о тех или иных особенностях документа. Следует заметить, что хотя в этой технологии применяется обработка событий, данный API не использует ни java.util.EventListener, ни java.util.EventObject.

Использование шаблонов

Observer (стр. 111). При организации обработки событий (использование этого шаблона) наиболее очевидно. На сегодняшний день создано множество различных событий, их обработчиков и источников. Каждый из них, в той или иной степени, является реализацией шаблона Observer, поскольку последний применяется в тех ситуациях, когда нужно отделить источник события от его обработчика. Тип события определяется его источником в тот момент, когда это событие возникает. Обработчик события предоставляет информацию о тех операциях, которые необходимо выполнить при возникновении события данного типа.

Adapter (стр. 156). Пакет java.awt.event содержит множество классов, в названии которых присутствует слово Adapter. Это позволяет сделать предположение о том, что такие классы являются различными вариантами реализации шаблона Adapter. Однако функции, выполняемые адаптерами событий, несколько отличаются от тех функций, которые должны выполнять классы, реализующие шаблон Adapter. Адаптеры событий используются для преобразования интерфейсов обработки событий в классы. Например, интерфейс MouseListener имеет соответствующий ему класс MouseAdapter, который реализует этот интерфейс и определяет ряд пустых методов. Поскольку классы адаптеров событий на самом деле не имеют функционального наполнения и не выполняют никаких преобразований между двумя разными интерфейсами, их нельзя рассматривать в качестве реализаций шаблона Adapter.

Factory Method (стр. 42). Класс AWTEventMulticaster содержит статические методы, обеспечивающие создание связного списка обработчиков событий определенного типа. Эти методы представлены методами регистрации (add) и ее отмены (remove) для каждого типа обработчиков событий, определенных в пакете java.awt.event, например: public static ActionListener add(ActionListener a, ActionListener b). Возвращаемое методом значение имеет тип того обработчика события, для которого выполняется регистрация. Метод реализован таким образом, чтобы возвращаемым объектом был зарегистрированный обработчик. Однако в большинстве случаев этот метод возвращает ссылку на новый экземпляр класса AWTEventMulticaster, который, в свою очередь, содержит ссылки на старый экземпляр этого класса и на новый экземпляр обработчика событий.

Composite (стр. 171). Класс `AWTEventMulticaster` обеспечивает создание связного списка обработчиков. При каждом добавлении нового обработчика экземпляр класса AWTEventMulticaster создает новый экземпляр этого же класса. Этот новый экземпляр получает ссылку на новый зарегистрированный обработчик и на уже имеющееся дерево объектов класса AWTEventMulticaster, а потом проделывает те же операции со своими двумя ссылками. Данный подход обеспечивает выполнение рекурсивных вызовов всего дерева объектов.

Chain of Responsibility (стр. 62). Класс AWTEventMulticaster перенаправляет все вызовы методам, определенным в интерфейсе тех обработчиков событий, ссылки на которые известны экземпляру класса. Иными словами, класс AWTEventMulticaster сам по себе практически не выполняет никакой работы. Он просто вызывает нужный метод у каждого из своих дочерних объектов. Если дочерний объект является обработчиком события AWT, он выполнит обработку события. Если же этот объект является экземпляром класса AWTEventMulticaster, он передаст вызов метода своим дочерним объектам.

Command (стр. 71). Класс AWTEventMulticaster имеет структуру, подобную структуре макрокоманды. Он имеет коллекцию других подобных объектов, которая в данном случае состоит из двух элементов. Один объект является терминальным (обработчик события), а другой— макрокомандой (экземпляр класса AWTEventMulticaster). Правда, вместо вызова метода выполнения, применяются вызовы других методов обработчиков событий.

Источник события, вызывающий метод, ничего не знает о структуре вызываемых классов (впрочем, он и не должен ничего о ней знать). В соответствии с шаблоном Command, источник события должен лишь вызвать метод обработчика события, обеспечивающий запуск последнего, а уж обработчик события начинает действовать подобно макрокоманде, отрабатывая поступивший запрос.

JavaBeans

Пакеты

`java.beans`, `java.beans.beancontext`
Используется в J2SE (начиная с JDK 1.1)

Обзор

Технология JavaBeans обеспечивает использование в языке Java стандартизованной модели, которая позволяет разрабатывать классы в виде компонентов. Компоненты характеризуются стандартным способом представления данных и функциональности, что облегчает их совместное использование разными разработчиками. В принципе, компонентная модель, подобная JavaBeans, позволяет разработчику воспользоваться кодом, созданным другим разработчиком, даже если он работает в другой компании, находящейся на другом континенте. При использовании компонентной модели функции разработчиков разделяются: одни являются программистами компонентов, другие — сборщиками компонентов, а третьи — сборщиками приложений. Собственно написанием программ занимаются только программисты, поскольку

сборщики компонентов и приложений применяют специальные инструменты разработки, которые позволяют им визуально манипулировать уже готовыми компонентами и объединять их в новые компоненты и целые приложения. На сленге технологии JavaBeans базовые компоненты и компоненты, полученные из базовых, называются кофейными зернами (beans), что и отражено в ее названии (стоит напомнить, что Java – это популярный сорт кофе). Технологию JavaBeans можно сделать визуальной, но сама по себе она не диктует требований визуальной реализации.

То, что компонентная модель является весьма удачной идеей, подтверждается тем фактом, что технология Java 2 Enterprise Edition построена на ее основе. Правда, с другой стороны, нужно отметить, что визуальное использование JavaBeans пока не нашло всеобщего признания.

Тем не менее, последнее обстоятельство вовсе не делает технологию JavaBeans не стоящей внимания. Когда она получила широкую поддержку, пришлось вносить изменения в язык Java. Старую модель управления событиями, построенную по иерархическому принципу, пришлось заменить более гибкой моделью, которая обеспечивает равномерное распределение ответственности за обработку событий. Эта новая модель, являющаяся архитектурной основой всей технологии JavaBeans, получила название "модель делегирования" (см. раздел "Обработка событий" на стр. 289).

Кроме того, особое внимание было уделено соблюдению соглашений об исходном коде, поскольку присвоение имен — это весьма важный аспект в технологии JavaBeans, используемый для получения информации о типах во время выполнения. Со временем все компоненты AWT стали соответствовать требованиям JavaBeans.

Технология JavaBeans была предложена для того, чтобы обеспечить поддержку событий, свойств (properties), получения информации о типах во время выполнения (introspection), настройки и долговременного хранения (persisting). Применяемая в настоящее время модель обработки событий позволяет разделить источники и обработчики событий. Каждый компонент, выполненный в соответствии с требованиями JavaBeans, может быть источником и (или) обработчиком событий. Такой компонент идентифицирует себя в качестве обработчика, реализуя соответствующий интерфейс и методы, определенные в этом интерфейсе. Если же ему нужно идентифицировать себя в качестве источника событий, компонент должен реализовать методы `addxxxxxxListener`и `removexxxxxxListener`.

Для того чтобы другие средства и приложения могли получить свойства компонента, он должен строго соблюдать соглашения о присвоении имен методам. Если компонент имеет методы, которые традиционно используются для получения имени и присвоения имени значения, например `String getNameO` и `void setName(String n)`, тогда другие приложения могут исходить из того, что данный компонент имеет свойство name типа `String`, хотя внутренне представление соответствующего поля может и отличаться от внешнего. Такое свойство может использоваться в таблицах свойств визуальных редакторов или в других приложениях. Технология JSP (Java Server Pages) основана именно на таком использовании компонентов JavaBeans: значения, возвращаемые формой HTML, устанавливаются в качестве значений свойств, к которым впоследствии можно обращаться для выполнения тех или иных операций.

Обычно все методы, объявленные общедоступными (public), экспортируются. Если поставщик компонента хочет ограничить перечень экспортируемых свойств, событий и методов, он может предоставить класс, реализующий интерфейс `BeanInfo`. Данный интерфейс определяет для других объектов методы, облегчающие выполне-

ние запросов на получение информации о доступных свойствах и событиях. Код, который определяет, какие методы являются экспортируемыми, должен находиться в реализации интерфейса `BeanInfo`.

Компонент может содержать собственный редактор свойств для новых типов данных, тем самым обеспечивая возможность своего подключения к визуальной компонентной среде. Такой редактор может либо работать со строковыми значениями, либо даже использовать компонент AWT, обеспечивающий редактирование с помощью графического пользовательского интерфейса.

Компонент JavaBeans должен также каким-то образом обеспечить поддержку своего долговременного хранения, поэтому ему нужно реализовать интерфейс `java.io.Serializable` или `java.io.Externalizable`. При сохранении компонента JavaBeans требуется сохранять и его внутренне состояние для того, чтобы при последующем восстановлении компонент смог продолжить работу с теми же данными, которые у него имелись в момент сохранения. Сериализованная версия компонента может даже рассматриваться как самостоятельный тип. Метод создания экземпляров компонентов JavaBeans `instantiate`, среди прочих параметров, получает имя компонента, передаваемое методу в виде строки. Этот метод пытается, прежде всего, найти файл с заданным именем и расширением `.ser`. Если такого файла нет, тогда он пытается найти класс с именем компонента и в случае успешного завершения поиска создает экземпляр этого класса.

Использование шаблонов

Factory Method (стр. 42). В технологии JavaBeans шаблон Factory Method используется для создания экземпляров компонентов с помощью метода Beans.`instantiate` и для обеспечения абстракции реального создания объекта. Объект, вызвавший метод, не видит никаких различий между экземпляром, извлеченным из файла после сериализации, и вновь созданным объектом. Это облегчает повторное использование компонентов JavaBeans. При настройке нужно лишь изменить некоторые свойства, чтобы компонент выглядел как компонент нового типа, затем выполнить сериализацию компонента и присвоить ему имя с расширением `.ser`.

Singleton (стр. 54). Для получения информации о компоненте JavaBeans приложения используют класс `Introspector`. Этот класс предоставляет информацию о поддерживаемых экземпляром компонента методах, событиях и свойствах. Он проходит по дереву наследования и извлекает открытую и скрытую информацию, которая используется при создании объекта Beanlnfo. Для обеспечения этой функциональности достаточно одного объекта класса `Introspector`. Чтобы избежать избыточности, которая может иметь место при использовании одного объекта, постоянно отвечающего на одни и те же запросы, в нем организуется кэширование информации.

Adapter (стр. 156). Шаблон Adapter занимает в спецификации JavaBeans особое место (в [JBS] он также упоминается, но под именем Adaptor). Основное назначение шаблона Adapter заключается в отделении источников событий от их обработчиков. Кроме того, реализация шаблона обычно выполняет одну или несколько из перечисленных задач.

- Реализация очереди для поступающих событий, что позволяет организовать поиск в том случае, если какое-то определенное событие окажется утерянным.
- Создание фильтра, который не позволяет всем событиям получать доступ к обработчику, а пропускает лишь те из них, которые соответствуют определенным критериям. Если, например, реализовать шаблон Adapter между компонентом Temperature, определяющим температуру, и компонентом Warning, выдающим предупреждение об изменении температуры, он обеспечит поступление к последнему компоненту лишь тех событий, которые несут информацию об изменении температуры более, чем на ОД градуса. Это позволит устранить ситуацию, при которой система постоянно выдает предупреждения из-за малейших колебаний температуры.
- Обеспечение демультиплексирования в системе. Класс может лишь один раз реализовать какой-то метод, определенный в интерфейсе. Если этот же объект должен обрабатывать одно и то же событие, поступающее от нескольких источников, но при этом реакция должна зависеть от источника, реализация метода обработки должна меняться для каждого нового источника. Традиционное решение заключается в использовании операторов switch, что приводит к неоправданному разрастанию объема исходного кода метода. В такой ситуации для обеспечения демультиплексирования можно использовать шаблон Adapter. Для этого необходимо создать объекты-адаптеры для каждого источника событий и зарегистрировать в соответствующих источниках именно эти объекты. При вызове метода обработки адаптер будет вызывать соответствующий метод реального обработчика, причем для каждого источника этот метод может быть уникальным. Теперь обработчик событий уже не должен определять, откуда поступило очередное событие, поскольку за него эту задачу решает адаптер.
- Обеспечение связи между источником событий и обработчиком. Эта функция шаблона актуальна в тех случаях, когда источник событий и реальный обработчик используют события разных типов. Данная функциональность является одним из ключевых признаков "истинного" адаптера, о чем можно прочитать в разделе "Adapter" на стр. 156.

Observer (стр. 111). Технология JavaBeans обеспечивает поддержку *связывающих* (bound) и *объединяющих* (constrained) свойств компонентов.

- Наличие у компонентов связывающих свойств означает, что такие компоненты могут связываться друг с другом, причем если изменяется какое-то из связывающих свойств, изменяются все взаимосвязанные компоненты. Эти свойства называются связывающими, поскольку они позволяют другим классам и объектам связать свое поведение с изменением соответствующего свойства.
- Объединяющие свойства подобны связывающим, но несколько отличаются от последних в реализации. При использовании объединяющих свойств в тех случаях, когда обработчик события обнаруживает изменение значения свойства, он может генерировать исключительную ситуацию PropertyVetoException. Обработчики событий должны регистрироваться с помощью метода addVetoableChangeListener (VetoableChangeListener listener), а их классы должны реализовывать интерфейс VetoableChangeListener. Компонент, в котором

произошло изменение свойства, вызывает метод vetoableChange зарегистрированного обработчика, передавая ему в качестве параметра экземпляр класса PropertyChangeEvent. Если возникнет исключительная ситуация PropertyVetoException, этот же компонент отменяет изменение и вызывает тот же самый обработчик, но уже с экземпляром класса PropertyVetoException, который восстанавливает прежнее значение свойства, тем самым выполняя откат ранее сделанного изменения.

AWT и Swing — API графического пользовательского интерфейса

Пакеты

Основными пакетами AWT являются `java.awt` и `java.awt.event`.

Кроме того, в состав AWT входят следующие пакеты: `java.awt.color`, `java.awt.datatransfer`, `java.awt.dnd`, `java.awt.font`, `java.awt.geom`, `java.awt.im`, `java.awt.im.spi`, `java.awt.image`, `java.awt.image.renderable`, `java.awt.print`.

Используется в J2SE (JDK 1.0, значительно расширен в JDK 1.2 и 1.3)

Основной пакет Swing называется `javax.swing`.

Кроме того, в состав Swing входят следующие пакеты: `javax.swing.border`, `javax.swing.colorchooser`, `javax.swing.event`, `javax.swing.filechooser`, `javax.swing.plaf`, `javax.swing.plaf.basic`, `javax.swing.plaf.metal`, `javax.swing.plaf.multi`, `javax.swing.table`, `javax.swing.text`, `javax.swing.text.html`, `javax.swing.text.html.parser`, `javax.swing.text.rtf`, `javax.swing.tree`, `javax.swing.undo`.

Используется в J2SE (начиная с JDK 1.2, расширен в JDK 1.3)

Обзор

Общие черты

Центральными концепциями AWT и Swing являются *компонент* (component), *контейнер* (container) и *диспетчер компоновки* (layout manager). Компонент— это базовый элемент графического пользовательского интерфейса, например, кнопка, надпись или текстовое поле. Контейнер— это также элемент графического пользовательского интерфейса, который отличается от компонента тем, что может содержать в себе другие элементы. Контейнеры являются организующими звеньями графических API Java. Они позволяют разработчикам группировать графические элементы в пространстве графического пользовательского интерфейса. Примером контейнера является окно, поскольку оно может содержать такие компоненты, как кнопки, флажки и т.п.

Диспетчеры компоновки не относятся к графическим элементам. Они представляют собой вспомогательные объекты, специализирующиеся на определении размера и расположения компонентов внутри контейнера. Контейнер делегирует задачу управления своим пространством связанному с ним диспетчеру компоновки, полно-

стью полагаясь на последнего в определении того, как и где размещать свои элементы пользовательского интерфейса.

Еще одним важным свойством AWT и Swing является наличие определенной модели обработки событий. В графических приложениях Java взаимодействие с пользователем представляется объектами событий, генерируемых компонентами. Например, если пользователь щелкнул большой красной кнопкой графического пользовательского интерфейса с надписью "Удалить историю", должно быть сгенерировано событие ActionEvent.

Не приведет ли это действие к неотвратимому уничтожению нашей цивилизации? К счастью, нет. Если нет соответствующего обработчика события, генерация события сама по себе не приводит к каким-либо реакциям со стороны программы. В данном примере нужен класс, реализующий интерфейс `ActionListener`, который был бы зарегистрирован в объекте кнопки с помощью вызова метода addActionListener. В таком случае событие ActionEvent было бы передано соответствующему обработчику ActionListener с помощью вызова метода последнего `actionPerformed`. Но поскольку на сегодняшний день еще никто не написал обработчик события для кнопки удаления истории, мы все можем спать спокойно.

Архитектурная модель AWT

API AWT (Abstract Window Toolkit) появился одновременно с самим языком программирования Java. Он основывается на простом наборе компонентов, позволяющем создавать базовые графические пользовательские интерфейсы. При создании приложения, использующего AWT, можно быть уверенным, что его компоненты будут выглядеть и вести себя одинаково на всех платформах, на которых работает JVM. Иными словами, один и тот же код, работая на платформе Solaris, будет генерировать графический пользовательский интерфейс в стиле Solaris, на платформе MacOS — в стиле Macintosh, и на платформе Wintel — в стиле Windows.

Этому факту имеется очень простое объяснение. Приложение AWT выглядит так, как если бы элементы его пользовательского интерфейса были реализованы для платформы, на которой выполняется это приложение, только потому, что эти элементы действительно реализуются для каждой конкретной платформы. Функциональность AWT базируется на концепции *пар* (peer). Каждый компонент AWT имеет парный ему класс, связанный с операционной системой, который выполняет всю основную работу по отражению компонента на экране. Классы, которые используют разработчики для создания графических компонентов в AWT, являются ничем иным, как обычными оболочками, скрывающими соответствующие пары. Например, программист использует класс AWT `java.awt.Button` для создания кнопки. С этим классом связан класс, реализующий интерфейс `java.awt.peer.ButtonPeer`, который и выполняет большую часть работы, связанной с отображением компонента на экране.

Из этого можно заключить, что должен быть какой-то способ для отслеживания зависимых от платформы пар внутри AWT. Такую возможность обеспечивает класс Toolkit. Внутри класса Toolkit имеется программный код, предназначенный для обеспечения связи с нижележащей операционной системой. Разработчики редко пользуются этим классом напрямую, но он занимает важную роль в AWT, поскольку именно он загружает графические библиотеки, создает пары объектов и управляет другими зависящими от платформы ресурсами (такими, как указатели мыши).

Базовые компоненты AWT нередко называют *тяжелыми компонентами* (heavyweight components), поскольку они напрямую связаны с нижележащей операционной системой через свои пары и при выполнении таких операций, как отображение на экране, всецело зависят от операционной системы. Решение об использовании пар, положенное в основу архитектуры AWT, предопределяет несколько важных последствий, которые нужно обязательно принимать во внимание при разработке приложений.

Достоинства

- Для такого API не нужно писать большой программный код, поскольку основную часть работы выполняет программный код нижележащей платформы.
- Графический пользовательский интерфейс выглядит и функционирует так, как если бы он был специально написан для той операционной системы, в которой он выполняется.

Недостатки

- В тех случаях, когда необходимо обеспечить настоящую независимость от платформы, при разработке компонентов приходится ориентироваться на наименьший общий знаменатель для всех платформ. Это означает, что графические пользовательские интерфейсы, созданные на основе AWT, выглядят весьма посредственно по сравнению с теми, что могли бы быть созданы при использовании других подходов.
- Если в графических компонентах операционной системы имеются какие-то отклонения от общепризнанных стандартов, приложение AWT наследует все эти отклонения вместе с нужной ему функциональностью.
- Поскольку во многих операциях, выполняемых компонентами графического пользовательского интерфейса, задействуются соответствующие им пары, работа приложения может замедляться, а также могут возникнуть проблемы при попытках его масштабирования.

Учитывая недостатки, связанные с использованием парных компонентов, в некоторых случаях разработчикам лучше применить прямое расширение одного или обоих базовых классов модели (Component и Container). Поскольку ни один их этих классов не имеет ориентированных на платформу пар, любой подкласс, созданный на их основе, будет наследовать базовую функциональность AWT, лишенную недостатков, присущих архитектуре, построенной на использовании пар. Конечно, это также имеет свои последствия — разработчики должны будут написать код, выполняющий отображение компонентов на экране.

Архитектурная модель Swing

Вся архитектура API Swing построена на концепции расширения функциональности класса AWT Container. К числу таких классов относится подавляющее большинство графических компонентов подкласса Swing JComponent, который, в свою очередь, является подклассом класса Container AWT. Это решение, положенное в основу архитектуры Swing, влечет за собой ряд последствий.

- API Swing построен на основе базовых классов API AWT, поэтому он наследует базовую модель AWT. Это означает, что приложение, построенное на использовании Swing, применяет такие же подходы при размещении компонентов (диспетчеры компоновки, контейнеры и компоненты) и обработке событий. Это также означает, что разработчики могут применять подобные приемы программирования при создании как приложений, основанных на Swing, так и приложений, основанных на AWT.
- Поскольку большинство компонентов Swing состоит из программного кода, написанного лишь с использованием языка Java без вспомогательных API, значительно повышается гибкость данного API. В частности, можно создать гораздо более широкий набор графических компонентов и сделать их более настраиваемыми, поскольку любой из них представляет собой просто некоторую совокупность точек на экране.
- Так как компоненты Swing являются подклассами класса Container, все они могут содержать в себе другие компоненты. В этом значительное отличие модели Swing от модели, принятой в AWT, в которой контейнерами могут быть лишь некоторые компоненты.

Конечно же, решение о создании одной архитектуры на основе другой само по себе не лишено определенных недостатков. Иерархия наследования классов Swing может быть слишком сложной, что порой препятствует пониманию того, какие функции выполняются на самом деле. Типичным примером является класс JButton, для которого иерархия наследования имеет следующий вид:

Object > Component > Container > JComponent > AbstractButton > JButton

Следует заметить, что в Swing осталось и несколько "тяжелых" компонентов. Они должны быть такими для того, чтобы взаимодействовать с нижележащей операционной системой. В частности, "тяжелыми" являются четыре оконных класса верхнего уровня, которые представляют собой подклассы соответствующих классов AWT (табл. 6.1).

Таблица 6.1. Классы AWT и Swing

Класс AWT	*Эквивалент Swing*
Applet	JApplet
Dialog	JDialog
Frame	JFrame
Window	JWindow

Эти четыре класса сохраняют вид и поведение, полученные от нижележащей операционной системы. Однако что касается остальных графических элементов, то может меняться не только их поведение, но и внешний вид. Компоненты Swing делегируют задачу своего отображения на экране связанному с ними классу пользовательского интерфейса, который может изменяться произвольно. В результате использования такого подхода Swing-приложение может, работая на платформе Windows, иметь графический пользовательский интерфейс в стиле Solaris. Эта возможность называется "подключаемые вид и поведение" (PLaF — pluggable look and feel).

Использование шаблонов

Общие вопросы

Observer (стр. 111). Имея сложную архитектуру, AWT и Swing не могут обойтись без использования шаблонов проектирования. Естественно, чаще всего в этих управляемых событиями API реализуется шаблон Observer, поскольку он предоставляет гибкий способ обмена информацией между объектами. Обе архитектуры используют шаблон Observer для управления обработкой событий. В обоих случаях графические компоненты представляют класс Observable, а программист должен написать класс Observer.

Composite и Chain of Responsibility (стр. 171 и стр. 62). Поскольку графические элементы AWT и Swing базируются на классах AWT Container и Component, эти API позволяют создание древовидных структур, представляющих объекты графического пользовательского интерфейса. Это означает, что в обоих API можно применять шаблоны Composite и Chain of Responsibility.

- Шаблон Composite реализован в нескольких методах контейнеров и компонентов, хотя нужно отметить, что он применяется несколько реже, чем того можно было ожидать. В частности, этот шаблон используется методом `list`, который предназначен для печати графического компонента в поток, а также методом readObject, назначение которого— **сериализация** компонента в поток. Еще несколько методов нельзя назвать истинной реализацией шаблона Composite, поскольку они вызывают различные методы для контейнеров и компонентов. Как известно, при истинной реализации шаблона Composite такого не должно быть — у класса Component должен быть один метод, который в случае необходимости, перекрывается другими классами.
- Шаблон Chain of Responsibility также реализован в нескольких методах. Как вы, очевидно, помните, реализация этого шаблона сопряжена с делегированием функциональности, которое чаще всего выполняется в пользу родительского объекта в древовидной структуре. Большинство методов класса Component, которые обращаются к стандартным свойствам компонента, используют этот шаблон. К таким методам относятся, например, getForeground, getBackground, getCursor и getLocale.

Использование шаблонов в AWT

Singleton (стр. 54). Класс `Toolkit` представляет собой интересный пример реализации шаблона проектирования Singleton. Этот класс применяет шаблон Singleton для генерации так называемого набора инструментов, используемого по умолчанию (default toolkit), а также для обеспечения того, чтобы соответствующий класс был доступен глобально. Используемый по умолчанию экземпляр класса Toolkit создается с помощью вызова статического метода `getDefaultToolkit` и обычно применяется разработчиками для получения набора инструментов, представленных классом Toolkit, при выполнении какой-то задачи наподобие создания задания печати. Поскольку класс `Toolkit` является абстрактным и требует переопределения для каждой конкретной операционной системы, вполне возможно, что какая-то конкретная реализация этого класса будет обладать необходимыми конструкторами и позволять соз-

дание других экземпляров класса (фактически, это и имеет место в реализации компании Sun для платформы Windows). Однако используемый по умолчанию набор инструментов всегда остается неизменным.

Bridge (стр. 164). Легко заметить, что архитектура AWT, основанная на использовании пар объектов, напоминает принцип организации шаблона Bridge, который разделяет компонент на две иерархии, одна из которых представляет уровень абстракции, а вторая — уровень реализации. Компоненты AWT в терминах шаблона Bridge представляют собой уровень абстракции (`Abstraction`), их пары— уровень реализации (`Implementor`), а пары для определенной операционной системы — уровень конкретной реализации (`ConcreteImplementor`). Однако есть и два небольших отличия от классической реализации шаблона Bridge.

- Многие классы компонентов на самом деле все же выполняют определенные функции самостоятельно, вместо того, чтобы делегировать их, как и все остальное, своей паре.
- Классы компонентов не уточняются. Это означает, что на практике отсутствует различие между классами `Abstraction` и `RefinedAbstraction`, как это должно быть при реализации классического шаблона Bridge.

Prototype (стр. 48). **В** AWT также можно встретить немало примеров реализации шаблона Prototype. Все они связаны с необходимостью получения копии экземпляра какого-то класса. Легко догадаться, что эти классы представляют ресурсы архитектуры AWT, которые, вероятно, используются приложением снова и снова: `Insets`, `GridBagConstraints`, `Area` и `PageFormat`.

Использование шаблонов в Swing

MVC (стр. 220). Можно сказать, что в API Swing лучше всего документирована реализация шаблона Model-View-Controller (MVC).

Практически все сложные элементы графического пользовательского интерфейса в Swing используют компонентную форму шаблона MVC. Имеется много веских причин в пользу применения этого шаблона, среди которых можно выделить следующие.

- Можно использовать единую нижележащую модель для управления множеством пар "представление — контроллер".
- Гораздо легче настраивать компонент, реализующий данный шаблон, поскольку программисту чаще всего нужно лишь модифицировать отдельные части функциональности компонента.

Следует отметить, что реализация шаблона MVC в API Swing выполнена очень тщательно. Функциональность модели, а также поведение контроллера, представлены интерфейсами. Управление элементами представлений осуществляется посредством иерархии классов пользовательского интерфейса, которая основывается на пакете `javax.swing.plaf`. Базовое поведение представления определяется серией абстрактных классов, которые впоследствии могут уточняться для обеспечения другого внешнего вида представления и другого поведения.

В качестве примера можно рассмотреть класс `JButton`, который используется для представления простой кнопки. Он связан с реализацией модели `ButtonModel`, с

представлением ButtonUI, а также, возможно, с одним или несколькими обработчиками событий, образующими контроллер.

Prototype (стр. 48). Подобно AWT, API Swing также имеет много вспомогательных классов, которые могут копироваться приложением, в связи с чем в этом API также присутствуют классы, реализующие шаблон Prototype: AbstractAction, SimpleAttributeSet, HTMLEditorKit, DefalutTreeSelectionModel.

Базовая среда Collections

Пакеты

java.util

Используется в J2SE (JDK 1.0, выделен в отдельный API начиная с JDK 1.2)

Обзор

Базовая среда (framework) Collections позволяет разработчикам использовать такие средства, как динамическое изменение размера, не прибегая к переписыванию исходного кода. С тех пор, как в языке Java впервые появились коллекции, API Collections претерпел ряд изменений. Базовая среда Collections предназначена для предоставления программистам более профессионального метода работы с коллекциями объектов.

Важно понимать, что на самом деле в языке Java имеется только один метод хранения групп элементов, заключающийся в использовании массивов (array). Можно сказать, что массивы — это базовые хранилища объектов. С их помощью в приложении образуются фиксированные наборы ссылок на объекты. Типы данных всех ссылок устанавливаются во время создания массивов, а их размер остается неизменным на протяжении всего существования массива в приложении.

Классы и интерфейсы, использовавшиеся в первых реализациях коллекций в JDK 1.0, бьии достаточно примитивными. В JDK были реализованы лишь три конкретных класса, основанных на двух типах коллекций.

- Vector и Stack. Коллекции, отсортированные по целочисленному индексу, который указывает на абсолютную позицию внутри коллекции.
- Hashtable. Коллекция, организованная по парам "ключ— значение". Ключ должен быть уникальным объектом, по которому осуществляется поиск значения. Значение может быть объектом любого класса и представляет собой элемент, который надлежит хранить в хеш-таблице. В любой структуре, подразумевающей хеширование, проверка уникальности ключа основывается на значении, возвращаемом методом hashCode (). Если два объекта при вызове их методов hashCode возвращают одно и то же значение, при сравнении их ключей они считаются одинаковыми.

Но современная базовая среда Collections состоит уже из десяти конкретных классов, построенных на основе многих интерфейсов и абстрактных классов. Более того, базовая среда предоставляет программисту возможность модифицировать функциональность от-

дельных коллекций. Программисты могут использовать класс `java.util.Collections` для расширения существующей коллекции, например, для синхронизации несинхронизированной коллекции.

В процессе работы группы по созданию базовой среды Collections для JDK 1.2 старые коллекции были переписаны заново, но при этом была обеспечена их совместимость с ранее созданными приложениями. Таким образом, новые "старые" коллекции стали органичной частью новой модели. Особенно впечатляет тот факт, что разработчикам удалось "вписать" старые классы в новую базовую среду, практически не внося каких-либо изменений в API. Команде разработчиков удалось обновить модель с минимальными изменениями в функциональности и без каких-либо потерь.

Конечно же, между исходными классами и классами, реализованными в новой модели, имеются некоторые различия. Прежние классы коллекций были созданы таким образом, чтобы работать с самого начала в отдельном потоке. Поэтому, просматривая документацию классов Hashcode, Stack и Vector, можно увидеть много методов синхронизации.

Сохранились ли методы синхронизации в каких-нибудь современных классах коллекций? Нет. Новая модель использует классы коллекций исключительно лишь для обеспечения базовой функциональности по хранению информации. Для работы с потоками необходимо использовать класс Collections, в котором определены соответствующие методы (пример приведен в листинге 6.1).

Листинг 6.1. Класс Collections и защищенный потоком вариант коллекций

```
1. List internalList = new LinkedList ();
2. List threadSafeList = Collections.synchronizedList(internalList);
```

Класс LinkedList обеспечивает хранение данных, а метод synchronizedList преобразует экземпляр этого класса в синхронизируемую коллекцию. Возможность использования синхронизации по мере надобности представляет программистам дополнительное удобство. Не вдаваясь в подробности, можно отметить, что программисту теперь не требуется использовать синхронизируемые коллекции (применение которых негативно сказывается на производительности), если это ему действительно не нужно.

Как уже говорилось, базовая среда Collections основывается на наборе интерфейсов. API имеет две основные функциональные иерархии. Первая базируется на интерфейсе Collection, который представляет коллекции, обеспечивающие простое хранение данных и применяющиеся для создания перекрестных ссылок на элементы. Вторая — на интерфейсе Map, который предназначен для описания коллекций, организованных по парам "ключ — значение".

Интерфейс `Collection` является родительским для трех подынтерфейсов.

- Set. Этот интерфейс определяет методы, доступные в любой коллекции без каких-либо дополнительных требований к соблюдению содержащимися в ней элементами определенной последовательности.
- List. Данный интерфейс определяет поведение коллекций, которые для определения позиции элемента в коллекции используют целочисленный индекс.

- `SortedSet`. Этот интерфейс используется для описания коллекций, в которых для организации элементов применяется "естественный порядок сортировки" (natural ordering). Элементы отсортированных множеств должны реализовывать интерфейс Comparable, в котором определен метод `compareTo`. Этот метод возвращает числовой результат, с помощью которого обеспечивается организация объектов отсортированного множества.

Интерфейс Map также имеет подынтерфейс SortedMap. Подобно SortedSet, этот интерфейс используется для получения схем с естественным порядком сортировки. Коллекции, реализующие этот интерфейс, должны каким-то образом обеспечивать сравнение ключей элементов, устанавливая в результате операции сравнения, является ли один из сравниваемых элементов меньше другого, равным ему или больше его. Ключи должны реализовывать интерфейс Comparable.

Использование шаблонов

В базовой среде Collections реализовано достаточно много шаблонов. В самой среде часто применяются шаблоны Prototype и Iterator, а класс Collections реализует шаблон Decorator.

Prototype (стр. 48). Шаблон Prototype использует один объект в качестве "заготовки" или базы для создания другого объекта. Учитывая назначение классов коллекций, не приходится удивляться, что все они поддерживают операцию *клонирования* (clone). В результате этой операции возвращается новая копия текущей коллекции. Все классы коллекций реализуют интерфейс Cloneable и возвращают при вызове метода clone так называемую *ограниченную копию* (shallow copy). Термин "ограниченная копия" в данном контексте означает, что в возвращаемом новом экземпляре класса коллекции все внутренние элементы представляют собой те же самые объекты, которые хранятся в исходной коллекции.

Iterator (стр. 87). Все реализации коллекций предоставляют возможность извлекать объекты из коллекций, последовательно перебирая элемент за элементом. Шаблон Iterator также обеспечивает выполнение простого перебора элементов коллекций. Интерфейс Collection содержит определение метода, названного iterator, а интерфейс List— метода `listIterator`. Эти методы возвращают реализации интерфейса, которые позволяют пользователям перемещаться по коллекциям. При этом метод iterator обеспечивает движение только вперед, а метод listIterator— в обоих направлениях.

Однако имена этих методов не должны никого вводить в заблуждение. На самом деле оба интерфейса реализуют классический шаблон Iterator не в полной мере, поскольку в них отсутствует полный перечень основных методов шаблона. В частности, ни один из интерфейсов не определяет в явном виде метод `first`. Однако целью такой, пусть и частичной, реализации является предоставление абстрактной функциональности перемещения на уровне реализации нижележащей коллекции. Поэтому главное назначение этих интерфейсов совпадает с назначением шаблона Iterator.

Классы коллекций для создания конкретных итераторов используют внутренние классы. При вызове метода iterator или listIterator коллекция создает объект внутреннего класса и возвращает его вызвавшему методу.

Decorator (стр. 180). Класс Collections использует шаблон Decorator для расширения функциональности коллекций. Такое расширение достигается за счет предоставления объектов, которые модифицируют поведение коллекций, не меняя их самих. Класс имеет три группы методов, генерирующих классы с дополнительными возможностями. В табл. 6.2 представлены эти группы методов и обеспечиваемая ими функциональность.

Таблица 6.2. Имена методов и обеспечиваемая ими функциональность

Начало имени метода	*Обеспечиваемая функциональность*
`singleton`	Создает постоянную одноэлементную коллекцию
synchroni zed	Создает коллекцию с синхронизируемыми методами
`unmodifiable`	Создает постоянную коллекцию

Вызов любого метода из этих групп приводит к созданию объекта, расширяющего возможности переданной ему в качестве параметра коллекции путем добавления к ней соответствующей функциональности.

Класс Collections на самом деле имеет набор внутренних классов, которые он использует для обеспечения этих дополнительных возможностей. Поэтому вызов метода `synchronizedCollection` будет приводить к генерации объекта-оболочки, закрывающего внутреннюю коллекцию для того, чтобы обеспечить синхронизацию методов, входящих в интерфейс Collections.

Примечание

Существует одно исключение из этого правила. Создание синхронизируемой коллекции не приводит к получению синхронизируемых методов итерации, поэтому разработчик должен обеспечить принудительную синхронизацию методов iterator и *`listIterator`***.**

На первый взгляд может показаться, что методы, имена которых начинаются со слова singleton, реализуют шаблон проектирования Singleton. Однако это не так, поскольку эти методы предназначены для решения совсем других задач. Эти методы ***не*** дают гарантий того, что в приложении будет существовать только один экземпляр объекта коллекции. Они лишь гарантируют, что в коллекции будет содержаться один-единственный объект. Любая попытка добавления в такую коллекцию элементов или удаления их из нее неизбежно приведет к генерации исключительной ситуации `UnsupportedOperationException`.

Ввод/вывод

Пакеты

`java.io`
Используется в J2SE (начиная с версии 1.0)

Обзор

Основное назначение API **ввода/вывода** языка Java состоит в том, чтобы обеспечить разработчикам возможность работы с потоками. Потоки (stream) обеспечивают в языке Java базовые средства **ввода/вывода**. Для осуществления записи в файл следует использовать поток. Если необходимо произвести чтение из стандартного ввода, также следует использовать поток. Даже если требуется выполнить запись по сети, ответ тот же — используйте поток.

Пакет `java.io` содержит четыре общих типа потоков, которые базируются на четырех абстрактных классах. Эти классы предоставляют функциональность, основанную на направлении потока (ввод или вывод), а также на том, является ли информация, выводимая в поток, байтовой или символьной (табл. 6.3).

Таблица 6.3. Типы потоков, представленных в пакете `java.io`

Низкий уровень (байты)		*Высокий уровень (символы)*	
Ввод	Вывод	Ввод	Вывод
`InputStream`	`OutputStream`	`Reader`	`Writer`

Все остальные потоки Java представляют собой подклассы одного из этих четырех классов, расширяя функциональность базового класса путем добавления дополнительных возможностей. Например, класс `FileWriter` относится к потокам класса `Writer` (вывод, запись символов) и добавляет возможность записи символов в файл на диске. Класс `DataInputStream` — это разновидность класса `InputStream` (ввод, чтение байтов), которая позволяет разработчикам считывать данные разного типа, таких как значения типа `int`, `float` или `boolean`.

Как создать поток, который бы объединял возможности различных потоков Java? Следует "сцепить" их друг с другом, используя концепцию *фильтрующих потоков* (**filter** stream). Фильтрующий поток можно присоединить к другому потоку, передав ссылку на этот поток в качестве параметра конструктору фильтрующего потока. После этого фильтр может добавить свою функциональность к связанному с ним потоку. Например, можно считывать строки из стандартного потока ввода, используя программный код, представленный в листинге 6.2.

Листинг 6.2. Потоки в Java

```
1. BufferReader readIn =
2.       new BufferedReader(new InputStreamReader(System.in));
3.     String textLine = readIn.readLine();
```

В данном примере ввод обрабатывается потоком `System.in` (ввод с клавиатуры), причем вводимая информация поступает в приложение в виде потока байтов. Добавив класс `InputStreamReader`, можно использовать получаемые байты для преобразования их в символы языка Java. Наконец, экземпляр класса `BufferedReader` помещает символы, полученные от экземпляра класса `InputStreamReader`, в буфер. Класс `BufferedReader` может обнаруживать символ конца строки и вытеснять хранящиеся в буфере символы в переменную типа `String`.

В самом пакете java.io определено лишь несколько классов потоков. Но поскольку разработчик может их объединять и применять фильтрующие потоки, в его распоряжении имеется гораздо больше функциональности, чем это может показаться на первый взгляд. Можно представить API ввода/вывода Java в виде трубопровода. При написании кода разработчик добавляет в приложение различные объекты ввода/вывода, или "трубы", соединяя их таким образом друг с другом. При добавлении каждой новой "трубы" происходит модификация потока, текущего по этой "трубе".

Использованиешаблонов

Очевидно, что при программировании ввода/вывода приходится часто менять потоки. С целью обеспечения такой возможности API ввода/вывода Java реализует поддержку одного из вариантов такого структурного шаблона проектирования, как Decorator (стр. 180).

Каждый из четырех абстрактных классов (InputStream, OutputStream, Reader и Writer) играет роль базы в декорируемой цепочке. Классы ввода/вывода, поддерживающие декорирование, имеют не менее одного конструктора, которому в качестве параметра передается ссылка на другой класс ввода/вывода. Пакет java.io содержит следующие категории классов ввода/вывода.

1. *Базовые классы ввода/вывода* (так называемые узловые потоки). Эти классы обслуживают конечные точки коммуникаций, поскольку всегда подключаются к каким-то конечным точкам ввода/вывода. Например, класс FileReader не является декоратором, поскольку он напрямую подключается к файлу.
2. *Парные потоки.* На первый взгляд, эти потоки могут реализовывать шаблон Decorator, поскольку они обладают способностью прикрепляться к другому потоку. Однако такие классы просто спроектированы таким образом, чтобы работать попарно — выход одного класса поступает на вход другого. Эти классы можно отнести к базовым классам ввода/вывода. Сами по себе они допускают декорирование, но их истинное назначение состоит в создании коммуникационного канала с другим потоком.
3. *Потоки класса* PipedWriterFilter. Фильтрующие потоки при организации цепочек используют шаблон Decorator. Для управления функциональностью декорирования классы фильтров используют следующие правила.
   - Фильтр декорирует класс, используя конструктор, который принимает в качестве параметра ссылку на один из четырех базовых классов ввода/вывода: InputStream, OutputStream, Reader или Writer.
   - Класс фильтра обычно декорирует класс такого же типа. Это означает, что фильтр класса InputStream будет декорировать поток, который также относится к классу InputStream. Важное исключение из этого правила составляют классы InputStreamReader и OutputStreamWriter, которые обеспечивают трансляцию байтов в символы.

Динамическая подгрузка классов

Пакеты

`java.lang.reflect`
Используется BJ2SE (начиная с JDK 1.1)

Обзор

API *динамической подгрузки* (reflection) позволяет подгружать неизвестные приложению классы, а также получать информацию о классах и объектах во время выполнения программы (introspection). Последняя возможность оказывается полезной в тех случаях, когда нужно обеспечить динамическое подключение новых классов к работающей программе. Используя динамическую **подгрузку**, можно подключить класс к программе, зная лишь его имя.

Хотя большая часть API находится в пакете `java.lang.reflect`, следует также в качестве части этого API рассматривать и класс `java.lang.Class`. Дело в том, что класс Class играет роль шлюза в функциональности динамической подгрузки. API динамической подгрузки определяет несколько классов, инкапсулирующих различные типы информации— например, такие классы, как Method, Field, Constructor и `Modifier`. Эти классы являются финализированными и создавать их экземпляры (за исключением класса `Modifier`) может **только** JVM.

Благодаря API динамической подгрузки, разработчик может динамически задействовать ранее неизвестный источник. Такая ситуация может возникнуть, например, если класс объекта неизвестен на этапе разработки. При этом в приложении вполне допустимы вызовы методов и конструкторов неизвестного класса (листинг 6.3), модификация его полей, создание новых массивов, а также доступ к этим массивам и модификация их элементов.

Листинг 6.3. Создание экземпляров неизвестного класса

```
1. Class class = Class.forName("имя класса");
2. Object o = class.newInstance();
```

В листинге 6.3 показано, как можно создать новый объект, зная лишь имя его класса. Метод `forName` получает в качестве параметра строку, содержащую имя класса, и пытается найти класс, файл которого имеет такое же имя. Найдя такой класс, он загружает его в загрузчик классов. Метод возвращает ссылку на экземпляр класса `Class`, который описывает загруженный класс. Когда вызывается метод newInstance вновь созданного объекта, этот метод создает новый экземпляр, используя конструктор без параметров. Строка, содержащая имя класса, может быть считана из файла, свойства или любого другого **источника**.

В тех ситуациях, когда объект получен из какого-либо источника и необходимо выяснить тип этого экземпляра, следует использовать код, подобный тому, который приведен в листинге 6.4. Метод getClass возвращает ссылку на экземпляр класса Class, который описывает класс объекта, а метод `getName` возвращает имя класса.

Это может оказаться полезным при отладке, когда тип полученного объекта неизвестен.

Листинг 6.4. Определение типа неизвестного объекта

```
1. Object unknownTypeObject = //получение какого-либо значения.
2. System.out.println(unknownTypeObject.getClass().getName());
```

При разработке J2SE версии 1.3 в API динамической подгрузки были внесены дополнения. Начиная с этой версии, разработчики получили возможность использования динамического прокси-класса. Этот класс создается во время выполнения программы и реализует несколько интерфейсов, которые также определяются во время выполнения. Класс Proxy, отвечающий за создание этого класса, кроме того, является суперклассом для всех остальных прокси-классов.

Метод создания экземпляра динамического прокси-класса имеет три параметра: ClassLoader, массив интерфейсов и InvocationHandler. Созданный экземпляр прокси-класса делегирует все вызовы методов классу InvocationHandler, который и отвечает за их выполнение. Для того чтобы сделать механизм вызова методов как можно более универсальным, класс InvocationHandler реализует лишь один метод invoke.

API динамической подгрузки предоставляет много интересных возможностей, но большинство разработчиков скорее всего ими никогда не воспользуются. Дело в том, что за гибкость, которую дают разработчику эти возможности, приходится расплачиваться производительностью. В соответствии с рекомендациями, изложенными в [BlochOl], следует отдавать предпочтение не динамической подгрузке, а использованию интерфейсов.

Однако это не означает, что динамическая подгрузка совсем не нужна. Некоторые приложения могут извлечь немалую пользу из применения динамической подгрузки, особенно те, которые базируются на технологии JavaBeans, а также на использовании инспекторов объектов, интерпретаторов и служб, подобных сериализации объектов, которым нужно получать информацию об объектах во время выполнения программы.

Использованиешаблонов

Factory Method (стр. 42). Шаблон Factory Method используется для создания экземпляров, не прибегая к прямому вызову конструктора. Он также позволяет разработчику воспользоваться возможностью возврата различных типов, а не только того класса, конструктор которого вызывается. Динамическая природа динамического прокси-класса предопределяет невозможность использования обычного конструктора, поскольку для того, чтобы вызвать такой конструктор, необходимо знать имя класса и (или) конструктора. Когда использование конструктора невозможно, для создания экземпляра допускается применение вызова статического метода. Класс Proxy имеет два метода, выполненных в соответствии с шаблоном Factory Method. Один из них (`getProxyClass`) возвращает ссылку на объект класса Class, который описывает динамический прокси-класс с указанным интерфейсом. Второй подобный метод (`newProxyInstance`) отличается от любого другого обычного метода. Он использует метод getProxyClass для получения экземпляра класса Class, а затем использует этот экземпляр для получения конструктора с его последующим вызовом.

Класс Array, который является классом-оболочкой для работы с массивами, реализует шаблон Factory Method по другим причинам. Создание массива требует точного знания типа элементов, а полученный в результате объект представляет собой массив элементов указанного типа, причем этот тип уже нельзя изменить. Это подобно тому, как происходит создание обычного массива, когда программист объявляет тип элементов массива в программе при его создании без возможности изменения этого типа по ходу выполнения программы. Объект, создаваемый методом, выполненным в соответствии с шаблоном Factory Method, не является экземпляром класса Array, что позволяет избежать использования конструктора, поскольку конструктор класса Array возвращает ссылку на экземпляр этого класса.

Facade (стр. 188). **В** API динамической подгрузки класс Class играет роль представителя реального подгружаемого класса, посредством которого можно узнать о последнем все его самые важные параметры. При этом можно получить доступ и к другим подгружаемым классам для более специализированной модификации, вызова методов или динамической подгрузки.

Proxy (стр. 209). Классы Field, Method и Constructor инкапсулируют концепции поля, метода и конструктора, соответственно. Это позволяет получать всю нужную информацию о классе, использующем механизмы динамической подгрузки. Например, применяя объект класса Method, который связан с конкретным методом класса, можно узнать, какие модификаторы были задекларированы для данного метода в объявлении класса, получить список типов обязательных параметров метода, а также узнать тип возвращаемого значения. Кроме того, с помощью объекта класса Method можно даже вызвать этот метод.

Объект класса Method в данном случае играет роль прокси-объекта по отношению к конкретному методу другого класса. То же самое относится и к другим упоминавшимся выше классам: Method, Field, Constructor и `Modifier` используются для создания **прокси-объектов** для представления частей других объектов.

Еще одной реализацией шаблона Proxy является класс `Proxy`. Специальный метод этого класса, выполненный в соответствии с шаблоном Factory Method, создает требуемый класс с нужной функциональностью. Получаемый подкласс класса Proxy реализует все заданные интерфейсы и методы. Реализация методов осуществляется таким образом, чтобы все вызовы передавались одному методу обработки, находящемуся в классе `InvocationHandler`.

Извне поведение динамического прокси-класса ничем не отличается от поведения обычного класса: все его интерфейсы реализуются и все методы вызываются. На самом же деле реальная реализация методов выполняется в классе InvocationHandler.

ТЕХНОЛОГИИ РАСПРЕДЕЛЕННОЙ ОБРАБОТКИ ДАННЫХ

Глава 7

Java Naming and Directory Interface (JNDI)

Пакеты

`javax.naming`, `javax.naming.directory`, `javax.naming.event`, `javax.naming.ldap`, `javax.naming.spi`

Используется в J2SE (JDK 1.3), J2EE (J2EE 1.2)

Обзор

API Java Naming and Directory Interface (JNDI) был разработан для обеспечения стандартизации доступа к службам поиска из программного кода, написанного на языке Java.

Это определение не очень четкое, но оно вполне соответствует назначению данного API. JNDI позволяет получить стандартизованный доступ к широкому спектру служб имен и каталогов. Использование этого API можно сравнить с владением всеми телефонными справочниками мира (причем носить эти справочники с собой не нужно, так как вам достаточно лишь знать телефон оператора, под рукой у которого находятся все эти справочники).

До создания JNDI разработчикам приходилось использовать специализированные API для организации доступа к различным службам. Например, для обмена информацией по технологии RMI группа развития информационных технологий должна была создать реализацию реестра RMI и обеспечить ее сопровождение, чтобы приложения могли определять, на каких серверах находятся удаленные объекты и что это за объекты. Для управления коммуникациями по технологии JDBC группе развития приходилось обеспечить механизм хранения информации, извлекаемой из удаленной базы данных. Для управления службой каталогов группа развития должна была сопровождать какую-либо схему, чтобы обеспечить навигацию по схеме каталога предприятия.

Итак, мы видим, что три различные службы поиска, построенные на трех различных технологиях скорее всего управляются тремя различными способами. Некоторым отделам по развитию информационных технологий, возможно, такая ситуация поначалу понравится, но впоследствии, столкнувшись с "суровыми буднями" разработки и сопровождения они быстро начинают понимать, что игра не стоит свеч.

JNDI консолидирует задачу управления поисковыми службами таким образом, чтобы приложения могли во всех случаях использовать только одну технологию. Более того, JNDI облегчает разделение конфигурации ресурсов и их поисковых характеристик, поэтому разработчику практически не приходится использовать в создаваемом им программном коде информацию, ориентированную на конкретную среду.

К тому же технология JNDI достаточно проста в использовании. Для того чтобы получить доступ к ресурсу, требуется лишь создать вспомогательный контекстный объект, с его помощью получить ресурс по его логическому имени, а затем преобразовать этот ресурс в объект требуемого типа, как показано в листинге 7.1.

Листинг 7.1. Использование технологии JNDI

```
1. javax.naming.InitialContext jndiCtx = new InitialContext();
2.   Object resource = jndiCtx.lookup("datasource");
3.   javax.sql.DataSource hal = (javax.sql.DataSource)resource;
```

После того, как JNDI вернет ссылку на ресурс, с не можно выполнять все обычные операции. В примере, показанном в листинге 7.1, эта ссылка используется для подключения к базе данных.

Приложения Java используют API JNMDI для получения доступа к диспетчеру имен JNDI. *Диспетчер имен* (Naming Manager), в свою очередь, использует не менее одной реализации интерфейса провайдера службы (SPI — service provider interface) JNDI для получения доступа к нижележащей службе. Такие службы могут быть предназначены для ведения структуры каталогов и хранения таких файлов, как, например, LDAP, NDS или NIS, либо с таким же успехом обеспечивать взаимодействие распределенных объектов, как в случае с RMI или CosNaming в технологии CORBA.

Все службы JNDI можно условно разделить на два типа. Первый тип — это *службы имен* (naming service), которые предоставляют механизм связи имени с объектом. Если объекту присвоено имя, его можно найти, обратившись к службе имен и указав это имя. Второй тип— *службы каталогов* (directory service), которые предоставляют механизм группового поиска информации в логической иерархии, подобной структуре каталога. В этой иерархии имена связаны с каталогами, которые, в свою очередь, могут содержать наборы атрибутов и значений. И для службы имен, и для службы каталогов имена JNDI представляют собой лишь некоторое условное обозначение, идентифицирующее объект в вычислительной среде. Точно так же, как в жизни имена представляют собой условные обозначения людей, так и в JNDI имена представляют сложные объекты.

В терминах JNDI *контекст* (context) представляет некоторую начальную точку, от которой разработчики начинают искать ресурс. Контекст содержит набор связей между именами и объектами, называемый *привязкой* (binding). Кроме того, контекст обеспечивает соблюдение соглашений о присвоении имен путем введения набора правил, устанавливающих, какие имена являются допустимыми.

API JNDI состоит из пяти пакетов, что позволяет четко определить соответствующую функциональность. В табл. 7.1 показано, каким образом выполнено разделение средств JNDI по пакетам.

Таблица 7.1. Пакеты JNDI и их назначение

Пакет	*Обеспечиваемая функциональность*
`javax.naming`	Базовая среда JNDI
`javax.naming.directory`	Расширения служб каталогов
`javax.naming.event`	Расширения обработки событий
`javax.naming.ldap`	Расширения для поддержки LDAP версии 3
`javax. naming.spi`	Интерфейс провайдера услуг. Представляет собой базовую модель, расширенную для обеспечения низкоуровневых служб, которые использует JNDI

Использование шаблонов

Некоторые ресурсы JNDI реализуют в той или иной степени перечисленные ниже шаблоны проектирования.

Factory Method (стр. 42). Реализация шаблона Factory Method позволяет получить стандартный метод, предназначенный для генерации какого-то продукта, поэтому реализацию данного шаблона обычно можно встретить в ресурсах JNDI, обеспечивающих создание соединения. Одной из лучших иллюстраций является класс JDBC DataSource, который часто хранится в JNDI, как часть решения J2EE. Класс DataSource представляет собой некоторый механизм, предназначенный для создания объектов класса Connection, которые позволяют клиентам обмениваться данными с реляционными системами управления базами данных (РСУБД).

Также довольно часто реализацию данного шаблона можно встретить в интерфейсе провайдера служб. Связанные с JNDI низкоуровневые реализации служб требуют наличия некоторого механизма, обеспечивающего соответствие вызовам API функциональности, реализованной в этих службах низкого уровня. Шаблон Factory Method реализован в таких классах, как DirObjectFactory, DirStateFactory, InitialContextFactory, `ObjectFactory` и StateFactory. Соответствующие методы перечисленных классов генерируют конкретные продукты, связанные с определенными типами служб.

HOPP (стр. 202). Многие технологии распределенных вычислений могут поддерживать распределение функциональности между локальными и удаленными объектами, поэтому неудивительно, что такие технологии, как RMI, CORBA и EJB реализуют шаблон проектирования HOPP. Конечно, API JNDI сам по себе не поддерживает и не реализует данный шаблон напрямую, но поскольку этот API обеспечивает канал связи для удаленных объектов, он определенным образом участвует в распределенных вычислениях.

Prototype (стр. 48). Архитектура JNDI содержит несколько реализаций шаблона Prototype, обеспечивающих поддержку дублирования объектов. В JNDI данный шаб-

лон в основном применяется в поиске ресурсов, когда требуется дублировать объекты, используемые для отслеживания имен ресурсов или атрибутов каталогов.

- `javax.naming`: `Reference`, `Name`, `CompoundName`, `CompositeName`;
- `javax.naming.directory`: `BasicAttribute`, `Attributes`, `Attribute`, `BasicAttributes`.

JDBC

Пакеты

`java.sql` (JDBC 2.1, основная часть API)
`javax.sql` (JDBC 2.0, дополнительный пакет)
Используется в J2SE (1.0, реструктурирован в JDBC 2.1 в версии 1.2)

Обзор

В наши дни базы данных встречаются повсюду — трудно даже представить какое-либо современное приложение масштаба предприятия, в котором бы не использовались базы данных в той или иной форме. Для того чтобы получить доступ к этим базам данных из программы, написанной на языке Java, необходимо использовать API Java для работы с базами данных (JDBC —Java Database Connectivity). JDBC — это базовая среда доступа к базам данных, построенная на стандартной реализации SQL, которая обеспечивает единый интерфейс верхнего уровня к модулям для работы с различными базами данных. JDBC предоставляет разработчику средства манипулирования данными, хранящимися в базе, которые не зависят от любых других СУБД.

Одной из проблем, возникающих при работе с этой средой, является то, что каждая база данных может использовать свой диалект языка SQL. Даже минимальные отличия такого диалекта от стандартного SQL порой может привести к весьма неприятным последствиям для разработчика, использующего JDBC. Однако базовая среда, по определению, должна быть как можно более гибкой и при этом как можно более простой. Поэтому, сознательно допуская возможную несовместимость с некоторыми диалектами SQL, разработчики JDBC остановились на том, что данный API должен состоять лишь из нескольких интерфейсов, причем каждый из интерфейсов имеет лишь несколько методов. Вследствие такого подхода полученный API JDBC оказался чрезвычайно простым в использовании.

Для того чтобы установить связь с базой данных, необходимо иметь в своем распоряжении драйвер, который понимает протокол соответствующей базы данных. Такой драйвер может входит в комплект поставки базы данных или может разрабатываться по специальному заказу независимым поставщиком. Драйвер должен содержать реализацию интерфейсов, специфичных для данного протокола.

Каждый драйвер имеет класс, реализующий интерфейс Driver. Когда класс загружен, он создает экземпляр самого себя и регистрирует его у диспетчера драйверов (объект класса DriverManager). Последний ведет список доступных ему драйверов. Когда запрашивается подключение к базе данных, диспетчер драйверов предпринимает

попытку найти подходящий драйвер. Для этого он проверяет свой список драйверов начиная с первого, указанного при создании объекта (это выполняется путем считывания значения свойства `jdbc.drivers`), до тех пор, пока не будет найден нужный драйвер или достигнут конец списка. Драйверы, которые загружаются во время выполнения, добавляются в конец списка, поэтому они также проверяются, но в последнюю очередь. После того, как нужный драйвер найден, метод getConnection диспетчера драйверов возвращает ссылку на экземпляр класса Connection. Объект класса Connection представляет сеанс связи с базой данных. Когда клиент вызывает метод `createStatement` объекта класса Connection, последний создает объект класса Statement, предназначенный для выполнения SQL-запросов к базе данных. Кроме того, могут создаваться и другие типы объектов класса Statement, предназначенные для более специализированных задач.

Объект класса Statement получает от клиента строку, содержащую оператор SQL, и выполняет запрос к базе данных, в ходе которого из нее извлекается информация или выполняется ее модификация. В зависимости от типа запроса, вызывается либо метод executeUpdate, либо метод executeQuery. При запросе информации (выполнении оператора SELECT) объект класса Statement возвращает объект класса `ResultSet`.

Объект класса ResultSet — это представление таблицы данных, в которой инкапсулированы результаты выполненного оператора SELECT. При каждом вызове метода next этого объекта, предназначенного для перебора элементов таблицы, курсор перемещается на следующую строку результатов. Когда курсор установлен на новую строку, можно извлечь значения столбцов, содержащиеся в данной строке. Для того чтобы прочитать остальные строки, необходимо вызвать метод next несколько раз.

Поскольку базы данных, как правило, очень быстро растут, результаты запросов также могут иметь достаточно большой объем. Для того чтобы избежать проблем, вызванных нехваткой памяти, объект класса ResultSet представляет не весь результат, а относительно небольшими пакетами (batch), помещая в каждый из них лишь несколько строк. Когда достигается конец текущего пакета, объект класса ResultSet запрашивает следующий пакет у базы данных. Для пользователя этот процесс остается прозрачным. Пример типичного использования объекта приведен в листинге 7.2.

Листинг 7.2. Извлечение результатов запроса из базы данных

```
1. Connection con = DriverManager.getConnection("адрес");
2. Statement stmt = con.createStatement();
3. String query = "SELECT * FROM students WHERE " +
4.     " iq GREATER THAN 140 AND sociallife='non-existent'";
5. ResultSet nerds = stmt.executeQuery(query);
6. while (nerds.next()) {
7.   String name = nerds.getString(1);
8.   int iq = nerds.getInt(2);
9.   //считывание строк из выходного набора и их обработка.
10.}
```

JDBC изначально поддерживает переход на следующую строку выходного набора, поскольку эта операция относится к перечню немногих базовых операций, обеспечиваемых всеми поддерживаемыми базами данных. Так как сегодня многие базы данных имеют более развитые управляющие структуры, подобные решения были реализованы и в JDBC 2.1. После выхода этой версии класс ResultSet поддерживает переме-

щение по выходному набору вперед и назад, а также относительные и абсолютные перемещения курсора. Кроме того, они поддерживают режим обновляемости. Последнее свойство означает, что при изменении данных нижележащей базы в момент выполнении запроса, выходной набор также изменяется для того, чтобы отразить внесенные изменения.

Самые передовые средства, подобные поддержке имен JNDI, API службы транзакций Java (JST – Java Transaction Service), буферизации соединений и наборам столбцов, входят в дополнительный пакет j avax . sql.

Использование шаблонов

Базовая среда JDBC состоит из множества взаимосвязанных объектов, которые определяются лишь своими интерфейсами. Это требует применения статических методов, чтобы можно было создавать экземпляры реализующие данные интерфейсы классов, ничего не зная о самих классах.

Abstract Factory (стр. 26). Классы, которые реализуют интерфейс Connection, используют шаблон Abstract Factory. Шаблон Abstract Factory обеспечивает классу необходимую ему гибкость. Использование этого шаблона позволяет написать приложение, которому ничего не будет известно о базе данных, с которой оно будет работать. Но поскольку приложению известен интерфейс Connection, оно сможет создавать запросы к базе данных и получать от нее результаты.

Это преимущество, заключающееся в высоком уровне гибкости, еще больше проявляется во время выполнения программы. Если реализующий класс определяется на этапе выполнения, приложение не придется переписывать при изменении базы данных или замены ее драйвера.

Factory Method (стр. 42). Как видно, в JDBC шаблон Factory Method используется повсеместно. API JDBC проектировался таким образом, чтобы обеспечить максимальную гибкость его использования, поэтому разработчики могут применять этот прикладной программный интерфейс, ничего не зная о реализующих его классах. Именно поэтому большая часть API JDBC состоит из интерфейсов. Но поскольку при использовании интерфейсов нельзя вызвать конструктор, для создания экземпляров классов применяется шаблон Factory Method. Именно так и происходит при вызове метода getConnection диспетчера драйверов— этот метод создает объект, реализующий интерфейс `Connection`, и возвращает ссылку на этот объект.

Шаблон Factory Method обеспечивает гибкий способ создания объектов без какой-либо информации об их реальном типе.

Bridge (стр. 164). Шаблон Bridge лежит в основе всего API JDBC. Действительно, JDBC представляет собой универсальный интерфейс ко многим базам данных. В этой коллекции интерфейсов, которая обеспечивает отделение реализации от клиента, просматриваются черты, присущие шаблону Bridge. Интерфейсы при этом являются функциональной абстракцией, которая отделена от реализации.

Благодаря такому подходу замена драйвера JDBC проходит для приложения, ориентированного на работу с JDBC, совершенно безболезненно. Драйвер, представляющий собой ничто иное, как реализацию интерфейсов JDBC, может менять внутреннюю логику работы, никак не влияя на то, как приложение использует данный драйвер.

RMI

Пакеты

`java.rmi`, `java.rmi.dgc`, `java.rmi.registry`, `java.rmi.server`, `java.rmi.activation`

Используется в J2SE

Обзор

RMI позволяет организовать в приложении взаимодействие с удаленными объектами, выполняя удаленные вызовы методов. Иными словами, эта технология дает возможность приложению вызывать методы объектов, которые находятся в другом адресном пространстве. Таким образом в Java обеспечивается такая же модель работы в распределенной среде, как и при организации коммуникаций на локальном уровне: Java-клиент может просто выполнить последовательность вызовов необходимых ему методов, не обращая внимания на то, являются ли они локальными или удаленными. При этом не требуется создавать объемного программного кода ни на стороне клиента, ни на стороне сервера, поскольку все управление удаленными коммуникациями при использовании технологии RMI полностью обеспечивается ее инфраструктурой.

Для определения методов, которые могут вызываться клиентом на сервере, RMI использует определенный интерфейс. Этот интерфейс должен расширять интерфейс `java.rmi.Remote`, который идентифицирует любой класс, использующий данный интерфейс, как участника RMI. Интерфейс используется как клиентом, так и сервером RMI. Сервер реализует интерфейс, предоставляя реальную функциональность. Клиент же использует интерфейс для идентификации тех методов, которые он может вызывать удаленно.

Для вызова методов сервера RMI клиент должен во время выполнения программы получить ссылку на объект удаленного сервера. При этом в распоряжении клиента должен оказаться объект, называемый *слепком* (stub), который позволит ему найти конкретный удаленный объект, работающий в другом адресном пространстве, и установить с ним связь. Существует два способа получения слепка, без которого невозможно осуществить удаленный вызов методов. Чаще всего для этой цели применяется служба имен, такая как JNDI или `rmiregistry`. Службы имен позволяют клиентам выполнять операции поиска удаленных объектов, а также получать слепки для осуществления удаленных коммуникаций. Второй способ получения слепка состоит в осуществлении удаленного вызова метода, который самостоятельно возвращает слепок вызвавшему его объекту.

Именно находящийся на стороне клиента слепок отвечает за осуществление коммуникаций с удаленным объектом, выполняющимся на стороне сервера. Он в значительной степени упрощает задачу клиента по осуществлению обмена информацией. Благодаря работе слепка в исходном коде клиента вызов удаленного метода выглядит точно так же, как и вызов обычного локального метода. Когда клиентское приложение осуществляет вызов удаленного метода, этот вызов поступает сначала слепку, который устанавливает связь с сервером, обменивается с ним информацией, а затем возвращает полученный результат клиенту.

Слепок осуществляет также маршализацию (marshall) параметров, используемых в вызываемых методах, а также демаршализацию (unmarshall) возвращаемого значения, передаваемого методом клиенту. Маршализация — это специальная форма сериализации, при которой для объекта создается оболочка класса `java.rmi.MarshalledObject`. Класс MarshalledObject содержит универсальный идентификатор ресурса (URI — Unified Resource Identifier) сериализованого объекта, по которому можно найти файл класса отправленного объекта, а также кодовую базу, представленную значением свойства `codebase`. Кодовая *база*— это строковое представление URL, по которому находится файл класса объекта.

Создать **RMI-приложение** довольно просто. Прежде всего необходимо найти удаленный интерфейс, т.е. интерфейс, который определяет, какие из доступных методов являются удаленными. Этот интерфейс расширяет интерфейс `java.rmi.Remote`, а все его методы должны объявляться таким образом, чтобы они при возникновении ошибок генерировали, как минимум, исключительную ситуацию `java.rmi.RemoteException`.

Удаленный объект реализует удаленный интерфейс в той степени, какую считает нужной. Когда необходимо переслать удаленный объект за пределы своего адресного пространства, на самом деле отсылается не сам удаленный объект, а его слепок. Для того чтобы обеспечить выполнение этой операции, удаленный объект должен каким-то образом "экспортировать" себя. Это означает, что JVM должна быть уведомлена о том, что этот объект является удаленным и что вести себя он будет соответствующим образом. **Неэкспортированный** удаленный объект при пересылке по сети ведет себя точно так же, как и любой другой (т.е. неудаленный) объект, копия которого пересылается по сети. Это обстоятельство может привести к непредвиденной ситуации. Существует два способа экспорта удаленного объекта. Первый состоит в расширении класса `java.rmi.server.UnicastRemoteObject`. В конструкторе этого класса объект является экспортируемым. Второй способ экспорта удаленного объекта заключается в явном вызове метода `exportObject` класса `UnicastRemoteObject`. Поэтому в данном случае удаленный объект может не расширять класс `UnicastRemoteObject`.

Для того чтобы удаленный объект был доступен другим объектам, такой объект должен зарегистрировать себя в службе имен.

Клиент находит нужный ему удаленный объект через службу имен. На свой запрос клиент получает объект, который, возможно, потребуется привести к заданному типу (интерфейс Remote), связанному с именем, которое было использовано в запросе. С этого момента клиент может использовать полученный объект для запуска его методов. Слепок переадресует вызовы удаленному объекту и будет ждать от них возвращаемого значения. При этом слепок может генерировать исключительные ситуации класса `java.rmi.RemoteException` для того, чтобы известить о возникших проблемах обмена информацией с удаленным объектом.

Слепок может генерироваться автоматически с помощью утилиты `rmic`.

Использование шаблонов

Abstract Factory (стр. 26). Шаблон Abstract Factory используется для создания семейств родственных продуктов. В RMI этот шаблон реализуется классом RMISocketFactory, методы которого создают сокеты, используемые при обмене данными по технологии RMI. В частности, класс `RMISocketFactory` содержит определения абстрактных методов, которые создают при **RMI-коммуникациях** сокеты на стороне

клиента и сервера. В каждой конкретной реализации JVM необходимо создать конкретный подкласс класса RMISocketFactory, на который возлагается задача обеспечения реальной функциональности для создания сокетов RMI.

Factory Method (стр. 42). Очень часто класс, реализующий шаблон Abstract Factory, использует для создания отдельных продуктов один или несколько методов, выполненных в соответствии с шаблоном Factory Method. Два интерфейса, реализуемые классом RMISocketFactory,— RMIClientSocketFactory и RMIServerSocketFactory, — определяют методы, которые отвечают за создание сокетов на стороне клиента и сервера, соответственно. Данные методы интерфейсов реализуются классом RMISocketFactory в виде абстрактных методов. Соответствующая функциональность, позволяющая воспользоваться технологией RMI во время работы программы, должна обеспечиваться подклассом класса RMISocketFactory.

Decorator (стр. 180). Шаблон Decorator позволяет расширить функциональность клиента за счет другого, специально созданного для этих целей объекта. Этот объект имеет такой же интерфейс, как и исходный объект. При выполнении большинства операций он не делает ничего другого, кроме как передачи управления исходному объекту, однако при этом он добавляет свою функциональность, которой у исходного объекта не было.

В технологии RMI шаблон Decorator используется при выполнении операций по сериализации объектов. Когда объект отсылается в другое адресное пространство, выполняется его маршализация; решение задачи отправки и получения копий объектов возложено на класс `java.rmi.MarshalledObject`. Для решения данной задачи класс MarshalledObject использует во время обмена данными подклассы классов `java.io.ObjectOutputStream` и `java.io.ObjectInputStream`. Эти специальные подклассы расширяют функциональность нижележащих классов `java.io.OutputStream` и `java.io.InputStream`, обеспечивая возможность проверки совпадения двух объектов класса MarshalledObject, даже если они находятся в адресных пространствах разных JVM.

Proxy (стр. 209). Слепок RMI, который используется клиентом для обмена информацией с объектом-сервером, представляет собой реализацию шаблона Proxy. Слепок реализует такой же удаленный интерфейс, что и удаленный объект на сервере, поэтому с точки зрения клиента RMI он выглядит точно так же, как представляемый им удаленный объект. Когда клиент вызывает метод удаленного объекта, слепок переадресует этот вызов реальному удаленному объекту, выполняющемуся на сервере. Достоинством этого подхода является то, что процесс обмена данными по сети скрыт от клиента. Это освобождает клиента от необходимости установления связи и управления сеансом, а также от участия в распределенной сборке мусора.

CORBA

Пакеты

JavaIDL: `org.omg.CORBA`, `org.omg.CORBA_2_3`, `org.omg.CORBA_2_3.portable`, `org.omg.CORBA.DynAnyPackage`, `org.omg.CORBAORBPackage`, `org.omg.CORBA.portable`, `org.omg.CORBATypeCodePackage`.

CosNaming: `org.omg.CosNaming`, `org.omg.CosNaming.NamingContextPackage`, `org.omg.SendingContext`.

RMI-IIOP: `javax.rmi.CORBA`, `org.omg.stub.java.rmi`
Используется в J2SE (JDK 1.2)

Обзор

Архитектура CORBA (Common Object Request Broker Architecture), построенная на использовании для обработки запросов специальных объектов-брокеров, представляет собой один из вариантов архитектуры системы с распределенным обменом данными между объектами. Проще говоря, CORBA— это технология, с помощью которой одно приложение может запрашивать службы другого приложения, вызывая удаленные методы объектов последнего.

CORBA великолепно справляется с задачей обеспечения взаимодействия различных приложений. Стандарт CORBA поддерживают множество современных языков программирования, а архитектура CORBA определена таким образом, чтобы клиент и сервер, написанные на разных CORBA-совместимых языках, могли взаимодействовать друг с другом, не владея информацией о деталях реализации удаленного визави.

Центральное место в архитектуре занимает брокер объектных запросов (ORB — Object Request Broker). Брокер ORB играет во всех распределенных коммуникациях роль маршрутизатора. Каждый клиент и каждый сервер системы не может работать, не обращаясь к ORB для обмена сообщениями.

Для того чтобы осуществлять взаимодействие с ORB, программный код каждого участника CORBA-совместимых коммуникаций должен быть написать таким образом, чтобы он соответствовал требованиям языка IDL (Interface Definition Language). IDL — это не зависящий от платформы язык, предназначенный для определения участниками CORBA-совместимых коммуникаций интерфейсов вызова. Именно этот язык играет ключевую роль в обеспечении совместимости клиент/серверных коммуникаций, участники которых разрабатываются с использованием разных языков программирования. Язык IDL позволяет описать формат контракта, которым обмениваются клиент и сервер. Обмен контрактом позволяет клиенту узнать, какие методы он может вызвать, причем независимо от того, на каком языке программирования были написаны эти методы.

Управление распределенными коммуникациями в базовой среде CORBA осуществляется по протоколу ПОР (Internet Inter-ORB Protocol). Этот низкоуровневый протокол был введен в последние версии спецификации CORBA для того, чтобы обеспечить осуществление распределенных коммуникаций на основе самой распространенной в настоящее время сети — Internet.

Java и CORBA

API CORBA для языка Java создавался таким образом, чтобы позволить программам, написанным на Java, взаимодействовать с приложениями, выполненными в соответствии со спецификациями CORBA, на нескольких уровнях. Поэтому в API Java реализованы три технологии, имеющие отношение к CORBA: JavaIDL, CosNaming и RMI-IIOP.

- **JavaIDL**. Технология JavaIDL представляет средства прямого доступа к технологии CORBA из программ, написанным на языке программирования Java. Соответствующий API и сопутствующий ему компилятор составляют инструментальный набор, позволяющий обеспечить соответствие между программным кодом на языке Java и интерфейсом IDL. С практической точки зрения это означает, что код Java может использовать данный API для того, чтобы функционировать согласно как модели клиента, так и модели сервера. Если же рассматривать вопрос шире, этот API позволяет Java-программам взаимодействовать с ORB и тем самым эффективно использовать все преимущества технологии CORBA.

- **CosNaming**. API CosNaming предоставляет программам, написанным на языке Java, возможность использования службы имен CORBA. Эта служба занимается исключительно задачей, вытекающей из ее названия, — обеспечивает соответствие объекта и некоторой строки, представляющей его имя.

  Сервер Java использует API CosNaming для выполнения регистрации своих удаленных объектов в службе имен, а клиент— для поиска и использования таких объектов. Это очень похоже на то, что выполняет интерфейс Java Naming and Directory Interface (JNDI), причем похоже настолько, что даже трудно указать на какие-то различия в функциях этих двух API. Однако с технологической точки зрения различия все же имеются. API CosNaming был создан раньше API JNDI для того, чтобы обеспечить совместимость приложений, написанных на языке Java, с технологией CORBA. В свою очередь, API JNDI был разработан таким образом, чтобы интегрировать в себя CosNaming в качестве одной из возможных служб имен. Это означает, что программист при разработке CORBA-совместимых приложений может как напрямую использовать API CosNaming, так и получить доступ к службе CosNaming, задействовав API JNDI.

- **RMI-IIOP**. Технология RMI-IIOP была добавлена в комплект технологий CORBA относительно недавно. Она позволяет RMI-совместимым программам осуществлять коммуникации, используя низкоуровневый протокол CORBA. В самой технологии RMI для транспортных целей используется протокол Java Remote Method Protocol (JRMP). Этот протокол был разработан на языке Java и исключительно для применения в программах, написанных на этом языке. Ни одна другая технология протокол JRMP не использует. Протокол же RMI-IIOP, являющийся коммуникационным протоколом CORBA, поддерживается очень многими языками программирования. Поэтому, вне всякого сомнения, имеет смысл, особенно учитывая тот факт, что язык Java все больше и больше становится универсальным языком программирования, использовать для обеспечения взаимодействия приложений именно протокол RMI-IIOP.

Нужно отдать должное разработчикам архитектуры RMI за то, что они постарались сделать незаметной замену протокола JRMP на протокол RMI-IIOP CORBA. Переход на RMI-IIOP был выполнен настолько удачно, что прикладным программистам не понадобилось вносить никаких изменений в ранее созданный программный код. Для того чтобы перевести RMI-совместимое приложение на использование протокола RMI-IIOP, достаточно просто перекомпилировать программу, ничего в ней не меняя.

Все три описанные выше технологии призваны привнести в язык программирования Java функциональность, определенную спецификациями стандарта CORBA.

При написании реальных приложений разработчики, использующие язык Java, как правило, задействуют лишь несколько классов из указанных в начале раздела пакетов. В частности, при работе с JavaIDL в подавляющем большинстве случаев используется только класс ORB, а при работе с CosNaming чаще всего задействуется не более полудесятка классов и интерфейсов. Что касается RMI-IIOP, то из классов, определенных в соответствующих пакетах, как правило, не используется ни один, поскольку программисты чаще всего обходятся одними лишь интерфейсами.

Возникает резонный вопрос: "А как же все остальные классы и интерфейсы CORBA? Для чего тогда они нужны?" Действительно, в состав всех API CORBA входит 143 класса и интерфейса, большинство из которых задействуются лишь вспомогательными службами. Так, например, происходит при генерации кода с помощью входящих в комплект CORBA утилит, таких как `idl2j ava`.

Использованиешаблонов

В API CORBA для языка Java реализованы лишь некоторые из основных шаблонов проектирования.

Singleton (стр. 54). Брокер ORB спроектирован таким образом, чтобы обеспечить в приложении единый коммуникационный центр CORBA для определенной виртуальной машины Java. Это означает, что в любом приложении Java имеется не более одного экземпляра ORB. Для обеспечения выполнения этого требования данный API реализует шаблон Singleton. Реализацию шаблона Singleton обеспечивает класс ORB, находящийся в пакете `org.omg.` CORBA. Метод `init` данного класса позволяет получить ресурс, соответствующий единому экземпляру брокера ORB для каждого приложения Java.

Factory Method (стр. 42). Класс ORB содержит много методов создания объектов, поскольку в технологии CORBA он является основным поставщиком ресурсов. Многие из этих методов соответствуют требованиям, выдвигаемым к программному коду, реализующему шаблон Factory Method, поскольку они должны генерировать объекты, спецификации которых могут гибко изменяться в момент создания этих объектов.

АРХИТЕКТУРЫ JINI И J2EE

Глава 8

Jini

Пакеты

Основные пакеты: net.jini.core.discovery, net.jini.core.entry, net.jini.core.event, net.jini.core.lease, net.jini.core.lookup, net.jini.core.event, net.jini.core.transaction, net.jini.core.transaction.server.

Утилиты и вспомогательные пакеты: net.jini.admin, net.jini.discovery, net.jini.entry, net.jini.event, net.jini.lease, net.jini.lookup, net.jini.lookup.entry, net.jini.space, com.sun.jini.admin, com.sun.jini.discovery, com.sun.jini.fiddler, com.sun.jini.lease, com.sun.jini.lease.landlord, com.sun.jini.lookup, com.sun.jini.lookup.entry, com.sun.jini.mahout, com.sun.jini.mahout.binder, com.sun.jini.mercury, com.sun.jini.norm, com.sun.jini.outrigger, com.sun.jini.reggie, com.sun.jini.start.

Используется в Jini 1.0

Обзор

Многие разработчики заявляют, что их "творения" являются воплощением *архитектуры, основанной на службах* (service-based architecture). Однако заявить — это далеко не всегда означает сделать. Технология Jini относится к редким исключениям, поскольку она действительно представляет собой образчик истинной реализации такой архитектуры. Это достигнуто путем создания понятных и простых в использовании интерфейсов.

Основное предположение, на котором зиждется Jini, состоит в том, что сеть является сущностью, неопределенной по своей природе, и потому ненадежной. Каждый, кому приходилось сталкиваться с обрывом связи на последних секундах загрузки объемного файла или с многократными безуспешными попытками подключения к перегруженному серверу, понимает, что это означает. О скорости соединения, с которой приходится работать в Internet подавляющему большинству пользователей, лучше вообще умолчать. Таким образом, сеть — это не просто некая абстрактная линия на красиво вычерченной диаграмме UML, а важная часть системы, обладающая вполне реальными характеристиками и особенностями. В сети любого типа может произойти неожиданное отключение пользователя во время работы, значительное падение пропускной способности сети или может появиться очень много ошибок. Постоянное присутствие некой вероятности отказа — это характеристика любой сети.

При создании технологии Jini разработчики исходили из того, что, во-первых, сеть существует и что, во-вторых, в ее работе могут возникать сбои. Такие предположения требуют, чтобы у пользователя или разработчика прикладных программ была возможность каким-то образом справиться с возникновением ошибки до того, как она нарушит нормальное функционирование приложения.

Архитектура Jini разрабатывалась для достижения следующих целей:

- обеспечить подключение к сети и отключение от нее без необходимости приостанавливать работу;
- обеспечить определение имеющихся аппаратных средств и установленного программного обеспечения;
- обеспечить естественную работу в сети;
- способствовать применению в приложениях архитектуры, основанной на службах;
- способствовать упрощению работы в сети.

Поисковые службы

Для того чтобы все заинтересованные стороны могли находить друг друга в системах, имеющих основанную на службах архитектуру, им нужно предоставить интерфейс, с помощью которого они могли бы узнать, как пользоваться теми или иными службами. Именно поэтому интерфейсы играют такую роль в службах технологии Jini. Пользователи службы ничего не знают о том классе, который реализует эта служба, поэтому они хотят иметь возможность находить любой подобный класс по заданному интерфейсу. Такой подход немного напоминает тот, что используется технологией RMI.

В технологии RMI для привязки имени к конкретному объекту используется служба имен. Однако использование имен — это прием с достаточно ограниченными возможностями. Например, если имя известно не точно, а лишь приблизительно, найти объект не удастся. Поэтому имеет смысл создать механизм, с помощью которого можно было бы найти нужный объект так, как это делается в телефонном справочнике — по разделу, соответствующему функциональности этого объекта. В языке Java функциональность определяется интерфейсом. В этой связи в технологии Jini были применены *поисковые службы* (lookup service), осуществляющие поиск по интерфейсу. Интерфейс поисковой службы определяется в пакете `net.jini.lookup.ServiceRegistrar`.

Класс ServiceRegistrar, по сути, представляет собой репозиторий служб Jini. Службы регистрируют себя в экземпляре класса ServiceRegistrar, играющего роль регистратора служб, а потребители услуг (т.е. пользователи служб) обращаются к нему при поиске нужных им служб. Регистратор служб поддерживает поисковые службы, основанные на базовом классе службы (net.jini.core.lookup.ServiceTemplate), что позволяет ему находить как любую произвольную службу, выполненную в соответствии с указанным базовым классом, так и коллекцию таких служб.

Еще один механизм, с помощью которого технология Jini обеспечивает гибкость и простоту использования, применяется для обнаружения поисковых служб. Хотя имеется возможность указать конкретную машину (или машины), на которой выполняется поисковая служба, такие службы можно находить динамически во время работы программы. Поэтому ни службе Jini, ни потребителю услуг не требуется знать, на какой конкретной машине работает служба.

Среди объектов Jini имеется предназначенный для пересылки по сети объект, который называется прокси-службой (service proxy), хотя этот объект может быть полноценной службой Jini.

Аренда

Одно из самых примечательных свойств Jini — это использование *аренды* (lease). Любой ресурс, которым пользуется или намерен воспользоваться потребитель услуг, выделяется последнему в аренду. Аренда — это извещение службы (владельца ресурса) о том, что пользователь службы намерен работать с данным конкретным ресурсом. Аренда имеет определенный срок действия, по истечении которого она перестает быть действительной, а пользователю требуется снова отправить запрос на аренду. Если служба хочет продолжить использование ресурса, она должна возобновить аренду. В том случае, если потребитель услуг не возобновляет аренду (ему больше не нужен ресурс, произошел сбой в сети и т.п.), ресурс может быть выделен в аренду другому потребителю.

Аренда не означает гарантированного использования. Потребитель услуг может постоянно отправлять запросы на аренду, но на каждый запрос ему будет приходить отказ. Дело в том, что служба может отменить аренду и освободить ресурс в любой момент, когда ей это необходимо. Конечно же, рекомендуется программировать ее работу таким образом, чтобы она выполняла подобные операции только в тех случаях, когда это действительно оправдано. Однако необходимо учитывать, что отказ в аренде может произойти даже в момент использования службы потребителем. Тем не менее, аренда не столь уж опасная операция, как это может показаться на первый взгляд. Как уже говорилось, если исходить из того, что сеть — это сама по себе ненадежная система, потребитель услуг должен быть готов к тому, что он в любой момент может столкнуться с отказом в доступе.

Предположим, что арендуемым ресурсом является служба доступа к телефонной линии, которая может обслуживать 10 линий связи. В обычных обстоятельствах, когда все 10 линий заняты, любая попытка воспользоваться телефоном будет отвергнута. Но при возникновении пожара потребность в телефоне получает наивысший приоритет. Поэтому служба доступа к телефонной линии должна быть запрограммирована таким образом, чтобы, получив сигнал от пожарной сигнализации, она прекращала действие аренды для, как минимум, одной линии, обеспечивала запрос ресурса (т.е. телефонной линии) и позволяла тем самым связаться с пожарной охраной.

Концепция аренды применяется очень часто. Когда служба регистрируется в поисковой службе, она, помимо прочего, получает в ответ экземпляр класса net.jini.core.lease.Lease. Служба использует этот объект для отмены или продления аренды, полученной от поисковой службы. Когда служба не продлевает аренду или отменяет ее, она удаляется из поисковой службы. Такой подход позволяет обеспечить соответствие информации, доступной поисковой службе, реальному положению вещей.

Распределенные события

Традиционная обработка событий, реализованная в языке программирования Java, в случае использования *распределенных событий* (distributed event) не может считаться приемлемой. Одна из причин — это наличие нескольких методов в интерфейсе обработчика, генерирующих удаленные исключительные ситуации RemoteException, что весьма нежелательно, когда обработчик находится в другом адресном пространстве. Более того, есть еще одна проблема, заключающаяся в наличии ссылки в объекте исключительной ситуации на источник событий. Если таким источником является компонент графического пользовательского интерфейса, такой компонент подлежит сериализации и отправке по сети. В чем же проблема? Да в том, что в результате по сети пересылается *весь* графический пользовательский интерфейс, поскольку все компоненты содержат ссылку на родительский объект. При программировании же распределенной обработки требуется, чтобы объекты были как можно меньшими. Наконец, третья причина состоит в том, что применяемая в настоящее время в языке Java модель обработки событий не поддерживает аренды.

Система обработки событий Jini изначально разрабатывалась с учетом ненадежности работы сети. Вместо нескольких интерфейсов обработчиков событий, в Jini представляется только один, net.jini.core.event.EventListener, у которого, в свою очередь, имеется лишь один метод обработки notify(net.jini.core.event.RemoteEvent re). Метод notify может генерировать исключительные ситуации двух классов: UnknownEventException и RemoteException.

Объекты класса RemoteException имеют минимальный размер, но при этом несут в себе всю информацию, необходимую обработчику. Класс RemoteException расширяет класс java.util.EventObject, поэтому в экземпляре класса RemoteException содержится ссылка на источник (удаленную службу), идентификатор события, возвращаемый объект и порядковый номер. *Идентификатор события* (event ID) обозначает то конкретное событие, для перехвата которого регистрируется обработчик. *Возвращаемый объект* (handback object) — это, как можно понять по его названию, объект, который предоставляется обработчиком события при регистрации и возвращается обработчику источником событий при возникновении соответствующего события (подробнее об этом говорится в подразделе "Использование шаблонов" данного раздела). Наконец, *порядковый номер* (sequence number) — это число, используемое для компенсации ненадежности сети. При сбоях события могут поступать в нарушенном порядке, а некоторые могут вообще пропадать. Поэтому по порядковому номеру обработчик может определить, в какой очередности обрабатывать события, поступившие к нему в произвольной последовательности.

Идентификатор события и порядковый номер неизвестны для обработчика до регистрации. Когда обработчик зарегистрирован, источник событий возвращает эк-

земпляр класса net.jini.core.event.EventRegistration, который содержит идентификатор события, текущий порядковый номер, объект класса Lease и ссылку на сам источник событий.

Использование шаблонов

HOPP (стр. 202). Когда клиент Jini (пытающаяся зарегистрироваться служба или потребитель услуг, обращающийся к поисковой службе, чтобы найти нужную ему службу) желает воспользоваться поисковой службой, он сначала находит один или более экземпляров такой службы с помощью протокола обнаружения. Для обнаружения поисковой службы клиент может воспользоваться классами LookupLocator, LookupDiscovery или LookupLocatorDiscovery.

Когда клиент обнаруживает поисковую службу, он сначала получает экземпляр класса ServiceRegistrar. Скорее всего объект класса ServiceRegistrar окажется удаленным объектом, поэтому полученный клиентом объект на самом деле будет прокси-объектом, обеспечивающим доступ к реальному удаленному объекту.

В Jini каждая служба имеет свой уникальный идентификатор. Поскольку поисковая служба является такой же службой Jini, как и все остальные, она также имеет свой уникальный идентификатор. Этот идентификатор присваивается службе лишь один раз, после чего она должна использовать его при каждом перезапуске. Хотя прокси-объект, играющий роль объекта класса ServiceRegistrar, переадресует большинство вызовов соответствующему удаленному объекту, не имеет смысла выполнять удаленный вызов, который сопряжен с высокими накладными расходами, если возвращаемый удаленным методом результат не изменяется. Таким образом, прокси-объект службы имеет специальный атрибут, в котором хранится значение идентификатора в поисковой службе.

Proxy (стр. 209). Термин "прокси-объект" толкуется в технологии Jini шире, чем в шаблоне проектирования Proxy. В соответствии с шаблоном Proxy, прокси-объект — это промежуточный объект, находящийся между клиентом и другим объектом. В Jini используется термин *прокси-объект службы* (service proxy), который обозначает часть службы, пересылаемую для выполнения на стороне потребителя услуг. Однако в отличие от промежуточного объекта в традиционном толковании шаблона Proxy, прокси-объект службы вполне может быть полнофункциональной службой, а процесс ее пересылки по сети на машину клиента остается совершенно прозрачным для потребителя услуг.

Observer (стр. 111). Когда потребитель услуг ищет определенную службу, может случиться так, что в момент отправки запроса данная служба по каким-то причинам недоступна. Повторять запросы до тех пор, пока служба не освободиться, — не самый эффективный способ решения проблемы. Значительно эффективнее, выполнив один раз попытку поиска, получить извещение о том, что в конфигурации служб произошли изменения. Для этого клиент использует метод notifyобъекта класса ServiceRegistrar. Так как для этого требуется принять участие в удаленной обработке событий, потребитель услуг предоставляет базовое определение (template) нужной ему службы, возвращаемый объект, экземпляр удаленного обработчика событий, с помощью которого он намерен получить извещение об изменениях, длительность аренды запроса, а также тип переходов (transition).

Java 2, Enterprise Edition (J2EE)

Обзор

Создание J2EE ознаменовало собой значительные изменения в применении языка Java, поскольку теперь его уже следует рассматривать не как набор API, а как полноценную базовую среду разработки. С концептуальной точки зрения, это определение как нельзя лучше подходит для J2EE — архитектурная базовая среда, используемая для создания приложений масштаба предприятия.

Начиная с выхода JDK 1.1 сильной стороной Java стали технологии распределенного программирования. Сокеты, JDBC, RMI, CORBA – все эти технологические новшества позволяли программистам создавать многоуровневые распределенные приложения.

Однако после появления J2EE область применения этих технологий подверглась существенному пересмотру. J2EE — это не еще одна форма обеспечения распределенных коммуникаций с помощью очередного API, а новая редакция языка, определяющая целостную модель, которая призвана обеспечить поддержку распределенной архитектуры.

Как можно заключить из названия, J2EE — это, действительно, не просто новая технология. Слово "редакция" упоминается в названии этой версии языка Java вполне оправдано, поскольку в J2EE представлена целая коллекция технологий Java, которые можно использовать совместно при реализации архитектурной модели. В данном случае модель предназначена для поддержки разработки и внедрения широкомасштабных распределенных приложений.

Основные концепции J2EE

В этом подразделе описаны основные концепции J2EE, а затем рассмотрены некоторые технологии, вошедшие в состав J2EE.

Уровни J2EE

Модель приложения масштаба предприятия в J2EE базируется на четырех логических модулях, или *уровнях* (tier).

- **Уровень клиента** (client tier). Обеспечивает пользовательский интерфейс. Уровень клиента может быть написан как на языке Java, так и на любом другом языке программирования. В J2EE уровень клиента обычно обменивается информацией с уровнем Web, используя протокол HTTP. В некоторых приложениях J2EE уровень клиента взаимодействует непосредственно с уровнями EJB или базы данных.
- **Уровень Web** (Web tier). Представлен броузером Web или отдельным клиентским приложением, обеспечивает функциональность масштаба предприятия для конечного пользователя. На уровне Web находится приложение Web, которое обеспечивает клиенту функциональность приложения в форме взаимосвязанного набора информационных ресурсов, передаваемых по протоколу

HTTP. Обычно эти ресурсы представлены документами, использующими технологии, подобные HTTP или XML.

- **Уровень сервера приложений,** или EJB (application server tier). Выполняет в объектно-ориентированной модели роль узла, представляя приложение в терминах взаимосвязанных объектов, имеющих отношение к прикладной области.
- **Уровень базы данных,** или сущности (database tier or persistence tier). Называется также уровнем информационной службы масштаба предприятия (Enterprise Information Service tier). Представляет в приложении все ресурсы предприятия, такие как базы данных, устаревшие приложения или системы совместной обработки данных предприятия.

Из всех четырех уровней больше всего привязаны к технологиям Java уровни Web и сервера приложений. Хотя остальные два уровня также могут использовать технологии Java, считается, что только сервер Web и сервер приложений по умолчанию построены на технологиях Java, если они входят в состав комплексного решения, полученного на основе J2EE.

Основные технические концепции

В центре модели J2EE находятся три взаимосвязанные технические концепции: *компоненты* (component), *контейнеры* (container) и *коннекторы* (connector). Компонент— базовая программная единица J2EE. Иными словами, J2EE подталкивает разработчика к созданию такой архитектуры приложения масштаба предприятия, которая была бы построена в виде набора взаимодействующих компонентов Java. В пользу такого решения говорят несколько доводов.

- Компоненты позволяют сделать программную систему масштаба предприятия более гибкой и расширяемой. При прочих равных факторах гораздо проще модифицировать систему, состоящую из набора компонентов, чем систему, имеющую монолитную архитектуру.
- Компоненты могут быть стандартизованными. Гораздо проще обеспечить соблюдение стандартных соглашений о написании программного кода для определенного типа компонента, чем для целого монолитного приложения либо для обширной библиотеки или базовой среды. Соблюдение стандартов облегчает задачу создания компонентов, которые можно подключать к другим системам, средам и прикладным моделям. Таким образом обеспечивается повторное использование отдельных частей архитектуры масштаба предприятия.
- Работа компонентов может основываться на службах. Мы всегда предпочитаем описывать приложения в терминах тех операций, которые они могут сделать для нас. Однако при переходе к объектно-ориентированной разработке вполне оправдано преобразование поведенческих характеристик в набор обеспечиваемых услуг. Если базовые строительные блоки естественным образом поддерживают концепцию служб, задача разработки приложения на их основе существенно упрощается.

Основные компонентные технологии

В J2EE реализованы три центральные компонентные технологии.

- Enterprise JavaBeans (EJB), которая используется на уровне сервера приложений
- JavaServer Pages (JSP)
- Сервлеты, которые вместе с JSP реализуются на уровне Web

Компоненты Java, по определению, требуют, чтобы какой-то программный объект занимался их размещением, управлял тем, когда они создаются и как вызываются их методы — иными словами, им нужен объект, который бы регулировал все, что происходит с компонентами на протяжении их жизненного цикла в системе. Именно эту задачу и решают *контейнеры*.

Контейнер обеспечивает компоненту J2EE наличие всех тех служб, которые нужны ему для выполнения своей задачи. Естественно, эти службы могут быть разными, в зависимости от того, о какой компонентной технологии идет речь. Например, для уровня Web фундаментальной задачей контейнера является трансляция Web-коммуникаций (запросов по протоколу HTTP) в вызовы методов компонентов, написанных на языке Java. С другой стороны, для уровня сервера приложений контейнер решает задачу управления коммуникациями между компонентами EJB посредством определенного протокола, работающего поверх протокола RMI-IIOP.

Контейнеры в приложениях масштаба предприятия не только обеспечивают требуемые службы на уровне отдельных компонентов, но и решают подобные задачи на уровне всего приложения в целом. Одной из главных причин, побуждающих к разработке систем масштаба предприятия, базирующихся на уже имеющихся программных продуктах, таких как Web-серверы, серверы приложений и базы данных, является желание воспользоваться готовыми возможностями, которые уже реализованы в этих продуктах и самостоятельная разработка которых потребует слишком много усилий. К таким возможностям относятся управление размещением данных, обеспечение безопасности, поддержка транзакций и др. В наши дни на разработку подобных служб "с нуля" обычно нет времени, поскольку к тому моменту, когда разработчики создадут работающие экземпляры подобных, довольно сложных приложений, система в целом может устареть и перестать соответствовать новым требованиям, выдвинутым жизнью.

Совместимые с J2EE продукты обеспечивают доступ к подобным службам для приложений J2EE и предоставляют при этом возможность настройки. Что еще лучше, они отделяют такие службы и их использование от самих компонентов, а это значительно повышает коэффициент повторного использования компонентов, позволяя применять их в широком диапазоне приложений. J2EE управляет конфигурацией служб, основанных на использовании контейнеров, посредством документов XML. Такие документы дают возможность указать на уровне компонентов, как контейнеры должны управлять безопасностью или хранением данных.

Коммуникации и коннекторные технологии

Системы масштаба предприятия, особенно те, которые были созданы за последние годы, развиваются в направлении объединения разных технологий, что влечет за собой проблему обеспечения управления коммуникациями между самыми разными приложениями. Например, нет ничего необычного в создании приложения масштаба

предприятия, которое должно взаимодействовать с базой данных, системой обмена сообщениями, почтовой системой и своим устаревшим предшественником, и которое будет работать до тех пор, пока все пользователи не переучатся для работы с новым приложением.

Конечно же, каждая технология стремится по-своему организовать взаимодействие с внешним миром, тем самым устанавливая свои коммуникационные стандарты. Даже в рамках одной технологии нередко можно встретить весьма разные стандарты на методы коммуникаций. Например, в базах данных практически каждая РСУБД использует собственный API. Исторически так сложилось, что в этом состоит основная проблема интеграции систем масштаба предприятия— приходится потратить очень много усилий, чтобы заставить все составляющие нормально взаимодействовать друг с другом.

В J2EE эта проблема решается путем создания универсальных коммуникационных технологий, получивших название коннекторных. С концептуальной точки зрения, коннекторы— это API языка Java, которые ограждают приложения J2EE от различий между конкретными коммуникационными моделями. Разработчики, использующие язык программирования Java, пишут программный код в соответствии со спецификациями API, который, в свою очередь, используется для связи с низкоуровневым коммуникационным слоем. Часто это означает, что посредством API обеспечивается подключение к модулю адаптера, который уже взаимодействует с другим приложением, системой или службой.

Как минимум, данный подход может обеспечить стандартизацию коммуникаций в определенной категории технологий, такой, как JavaMail для почтовых служб или JDBC для обмена информацией с реляционными базами данных. Используя этот подход, разработчики, выбравшие в качестве среды J2EE, могут применять одни и те же приемы программирования при написании базы данных, создавая тем самым нейтральный (с точки зрения реализации) код JDBC. Точное определение того, как JDBC взаимодействует с соответствующей РСУБД, обеспечивается драйвером JDBC, но API для разработчика всегда остается неизменным, независимо от того, какой уровень находится под этим API.

Основные ресурсы J2EE

Многие разработчики, приступающие к использованию J2EE, поначалу пытаются рассматривать эту среду, размышляя традиционными категориями Java, — как набор API, взаимодействующих друг с другом. Однако, помимо API, компания Sun разработала много других стандартных ресурсов, которые могут поколебать такой взгляд на J2EE.

- **Спецификации.** Спецификации — это связанные с основными технологиями J2EE документы, которые описывают, как технологии должны работать в среде предприятия и как разработчики могут использовать их в этом контексте.
- **Пример реализации.** Подобно многим технологиям Java, для J2EE имеется пример реализации, который распространяется отдельно по запросу разработчика или поставщика J2EE. Наличие такого примера позволяет начать работу над приложением масштаба предприятия, используя его в качестве отправной точки. В примере реализуются основные API J2EE и содержатся контейнер Web, контейнер EJB, реляционная база данных, а также различные инструменты для тестирования и внедрения приложений, основанных на J2EE.

- **Советы по J2EE.** Руководства по созданию приложений масштаба предприятия, а также готовые решения, которые можно использовать в разработке приложений на основе J2EE.
- **Пример приложения.** В качестве примера приложения, созданного на основе J2EE, используется Java Pet Store. Это приложение, снабженное исходным программным кодом, призвано продемонстрировать приемы использования архитектуры на практике.
- **Тестирование совместимости.** Поставщик, который желает разрабатывать совместимые с J2EE серверы, может воспользоваться пакетом Compatibility Test Suite для того, чтобы проверить, насколько его приложения соответствуют спецификации.

Шаблоны компонентов

Тремя основными компонентными технологиями J2EE являются EJB, JSP и сервлеты. Хотя в число основных можно было бы отнести и технологию аплетов, которая может применяться в приложениях Web, работающих на уровне клиента, все же она не занимает столь важного места в базовой модели J2EE, как остальные три технологии. В последующих разделах будут рассмотрены три основные компонентные API, а также описаны реализуемые ими шаблоны проектирования.

Сервлеты и JSP

Пакеты

`javax.servlet`, `javax.servlet.http`, `javax.servlet.jsp`, `javax.servlet.jsp.taglib`.

Используются в J2EE (начиная с версии 1.2)

Обзор

API сервлетов (servlet) обеспечивает в J2EE одну из двух компонентных технологий Web. Общая модель этого API достаточно проста и базируется на двух пакетах, в которых представлена вся основная функциональность:

- `javax.servlet` — собственно модель сервлетов;
- `javax.servlet.http` — адаптация модели сервлетов для работы с протоколами HTTP и HTTPS.

Базовая архитектурная модель сервлетов не использует никаких протоколов, работающих по принципу "запрос/отклик". Она основывается лишь на предположении о том, что в системе реализована поддержка стека протоколов TCP/IP. Естественно, построенные в соответствии с этой моделью ресурсы, с которыми работают сервлеты, достаточно просты и состоят лишь из того, что можно получить в сети, о которой неизвестно ничего, кроме низкоуровневых протоколов.

С практической точки зрения, большая часть функциональности сервлетов сосредоточена в пакете `javax.servlet.http`. Это позволяет адаптировать архитектуру сервлетов к миру Web, обеспечивая реализацию модели, которая основывается на протоколах HTTP и HTTPS.

Основные элементы API

Структурную основу сервлетов образуют три элемента API: интерфейс `Servlet`, класс GenericServlet и класс HttpServlet.

- Интерфейс Servlet определяет, помимо прочего, методы, управляющие жизненным циклом сервлетов: `init`, `destroy` и `service`.
- GenericServlet — это абстрактный класс, обеспечивающий базовую реализацию интерфейса, за исключением метода `service`. Поскольку любой класс сервлета должен, как минимум, по-своему определять определенное поведение в ответ на запрос клиента, этот метод в классе GenericServlet остается нереализованным.
- Класс HttpServlet также является абстрактным, но все его методы реализованы. Этот класс предоставляет метод `service`, за которым стоят семь методов вида doXxx. Данные методы соответствуют большинству используемых в настоящее время типам запросов HTTP: doGet, doPost, doPut, doHead, `doOptions`, `doTrace` и doDelete.

Жизненный цикл

Жизненный цикл сервлета, как и всякого компонента, зависит от управления контейнером Web. Контейнер руководит вызовами множества методов API сервлетов. В приведенном ниже списке перечислены основные методы сервлета, вызываемые контейнером на протяжении жизненного цикла последнего, а также показан порядок, в котором они вызываются.

- Создание одного или нескольких объектов сервлетов.
- Инициализация сервлета посредством вызова метода `init`.
- Обработка клиентских запросов с использованием для запуска служебных методов сервлета обработчиков потоков. За этими методами стоят методы вида doXxx, которые соответствуют основным командам протокола HTTP (doGet, doPut, doPost и т.д.).
- Вызов метода `destroy`.
- Удаление объектов сервлетов.

JavaServer Pages

JavaServer Pages — это технология, позволяющая специалистам по HTML и сценариям разрабатывать динамические страницы HTML. В таких страницах используются *дескрипторы* (tag) JSP, подобные дескрипторам HTML, которые могут содержать или вызывать программный код, написанный на языке Java. Элементы страницы HTML, содержащие дескрипторы JSP, преобразуются в сервлеты Java.

Когда Web-сервер встречает в коде HTML вызов JSP, он сначала передает этот вызов интерпретатору. Интерпретатор преобразует вызываемый файл в файл исходного текста на языке Java, транслируя все дескрипторы JSP в код Java. Затем выполняется компиляция файла с исходным текстом на языке Java в структурный эквивалент сервлета. Фактически когда запускается код JSP, его жизненный цикл в точности соответствует жизненному циклу сервлета, поэтому контейнер Web может управлять кодом JSP точно так же, как и сервлетом.

Сервлеты и JSP используют одну и ту же архитектурную модель, а также немало одинаковых классов. Некоторые модификации, предназначенные для поддержки модели JSP, а также вспомогательные классы, используемые этой технологией, собраны в пакете `javax.servlet.jsp`. Интерфейсы JspPage и HttpJspPage содержат определение базовой функциональной модели JSP. Но несмотря на то, что структуру JSP задают различные интерфейсы, типы используемых методов остаются неизменными—методы `jspInit`, `jspDestroy` и `jspService` являются аналогами методов сервлетов `init`, `destroy` и `service`.

API JSP также определяет набор классов, которые позволяют создавать библиотеки дескрипторов, определяемых пользователем. Эта технология позволяет расширять JSP с помощью программного кода, написанного на языке Java. Программисту нужно создать лишь три компонента. Первый компонент — это класс Java, содержащий реализуемую функциональность Java. Второй— файл XML, который функционирует как дескриптор внедрения класса, поскольку в этом файле содержится вся основная информация о классе, такая, например, как доступные для использования атрибуты. Третий компонент определяет использование самого JSP. В нем разработчик сценария JSP указывает файл класса Java, файл XML и любую другую информацию о реализации, такую, например, как информацию об атрибутах.

Использование шаблонов

Template Method (стр. 146). API сервлетов в нескольких местах очень тесно приближается к тому, что можно было бы назвать примером реализации шаблона Template Method. К сожалению, в одной, но ключевой области он не дотягивает до такого определения — в действительности этот API предоставляет используемую по умолчанию реализацию методов. Методы класса `HttpServlet`, которые ближе всего подошли к реализации данного шаблона, перечислены в следующем списке.

- Метод `service`, за которым стоят методы вида doXxx. Обычно разработчикам требуется перекрыть один или несколько методов doXxx, чтобы реализовать требуемую функциональность сервлета.
- Метод `init(ServletConfig)`, вызывающий метод `init`. Разработчики должны перекрывать метод init, если требуется обеспечить выполнение определенных операций при инициализации.

Session (стр. 231). API сервлетов предоставляет два механизма, обеспечивающих поддержку шаблона Session, — класс Cookie и интерфейс `HttpSession`. Как видно из их названий, класс Cookie представляет так называемые cookie-файлы HTTP, которые позволяют API сервлетов использовать информацию о сеансе, хранящуюся на стороне клиента Web. Интерфейс HttpSession применяется для создания хранилища информации о сеансе на стороне сервера.

Observer (стр. 111). Подобно многим другим API языка Java, API сервлетов использует модель обработки событий. Из этого следует, что таким образом сервлеты реализуют шаблон Observer. Этот шаблон используется сервлетами для извещения обработчиков событий об изменениях объектов класса HttpSession и ServletContext. В табл. 8.1 представлены интерфейсы обработчиков событий, которые можно использовать для создания наблюдающих объектов в приложениях Web.

Таблица 8.1. Интерфейсы и их назначение

Интерфейс	*Назначение*
`HttpSessionActivationListener`	Активизация или деактивизация сеанса
`HttpSessionAttributesListener`	Изменение в атрибутах сеанса
`HttpSessionBindingListener`	Извещение объекта о том, что он будет привязан к сеансу или отключен от него
`HttpSessionListener`	Создание или уничтожение сеанса
`ServletContextAttributesListener`	Изменения в атрибутах контекста сервлета
`ServletContextListener`	Создание или уничтожение сервлета

Enterprise JavaBeans

Пакеты

`javax.ejb`, `javax.ejb.spi`
Используется в J2EE (начиная с версии 1.2)

Обзор

По сути, технология Enterprise JavaBeans (EJB) является сердцевиной архитектуры J2EE. С помощью компонентов EJB в J2EE представляется основная бизнес-модель, определяемая в терминах набора взаимодействующих компонентов. Компоненты EJB используются и для реализации бизнес-правил, инкапсуляции бизнес-логики, а также для преобразования бизнес-модели в приложение масштаба предприятия.

В EJB имеется три фундаментальные категории.

- **Компоненты сеансов** (Session Bean). Компоненты EJB, которые непосредственно обеспечивают поддержку бизнес-логики. Компоненты сеансов могут быть как *ориентированными на состояние* (Stateful), так и *не учитывающими эти состояния* (Stateless). Компонент Stateful Session Bean связан с конкретным клиентским сеансом, тогда как компонент Stateless Session Bean представляет общий бизнес-ресурс, не зависящий от какого-то ни было конкретного клиента, обращающегося к нему.

- **Компоненты** сущностей (Entity Bean). Предназначены для обеспечения непосредственной связи с СУБД или другим постоянным хранилищем данных, представляя данные в "объектной" форме.
- **Компоненты, управляемые событиями** (Message-driven Bean). Функционируют как получатели асинхронных сообщений-извещений, поступающих от технологии обмена сообщениями в Java (JMS – Java Messaging System). Такие компоненты могут получать сообщения и каким-то образом реагировать на них, используя API JMS во время работы приложения J2EE.

Для того чтобы создать компонент EJB, нужно написать три кодовых элемента Java.

- Интерфейс `Home`, предназначенный для управления жизненным циклом компонентов EJB (должен содержать методы `create`, `locate` и remove).
- Интерфейс Remote, используемый для определения бизнес-методов.
- Реализация компонента масштаба предприятия (Enterprise Bean).

Кроме того, нужно написать документ XML, называемый дескриптором внедрения (deployment descriptor) и содержащий описание деталей управления компонентом Enterprise Bean (информация о настройке, администрировании и управлении ресурсами).

Компоненты EJB зависят от своих контейнеров в значительно большей степени, чем в любой другой компонентной технологии J2EE. Следует отметить, что компоненты EJB очень нуждаются в своих контейнерах, т.к. последние создают их, вызывают и регулируют все аспекты их жизненного цикла.

Интересно заметить, что вся архитектура EJB построена на интерфейсах. Единственные классы, которые определены на уровне модели, — это классы исключительных ситуаций. Этот факт иллюстрирует еще в большей степени зависимость компонентов EJB от своих контейнеров, поскольку одной из задач контейнера является генерация вспомогательного кода, который реализовывал бы интерфейсы `Home` и Remote и привязывал бы их к нижележащему компоненту Enterprise Bean.

Внедрение компонентов Enterprise Bean требует создания классов, которые реализуют интерфейсы `Home` и Remote. Это, в свою очередь, требует разработки программного кода, обеспечивающего соответствие между вызовами методов, определенных в интерфейсах, и вызовами реальных методов, реализованных в нижележащих компонентах Enterprise Bean. В некоторых случаях при этом требуется создать дополнительный управляющий программный код, основывающийся на информации, предоставленной дескриптором внедрения.

Нередко программистам, имеющим опыт разработки программного обеспечения среднего уровня (middleware), требуется время, чтобы оставить все наработанные ими приемы программирования и свыкнуться с мыслью о том, что обо всех дополнительных службах, требуемых компонентами EJB, позаботится контейнер. Занимаясь всю жизнь вопросами обеспечения увязки совместной работы компонентов, как-то сложно сразу воспринять концепцию автоматической генерации большого вспомогательного кода.

Использованиешаблонов

Общие вопросы

В технологии Enterprise JavaBeans реализуется несколько шаблонов, но многие из них предоставляются контейнерами, которые реализованы в базовой среде более низкого уровня в качестве одной из подзадач получения функционального и интегрированного компонента EJB.

HOPP (стр. 202). Клиенты никогда не взаимодействуют с компонентами Enterprise Bean напрямую. Они всегда передают вызовы методов либо интерфейсу `Home`, либо интерфейсу Remote, которые затем и переадресуют вызов компоненту EJB. Более того, можно сказать, что клиенты не могут напрямую взаимодействовать даже с классами, реализующими интерфейсы `Home` и Remote. В действительности они взаимодействуют со слепками, которые в свою очередь осуществляют обмен данными с соответствующими реализующими классами, находящимися на стороне сервера. Все вышесказанное демонстрирует использование шаблона HOPP, который применяется во всех коммуникациях компонентов EJB.

Factory Method (стр. 42). Компоненты EJB в качестве отправной точки для контакта с ресурсом EJB используют интерфейс `Home`. Клиенты, которыми могут быть сервлеты, JSP или другие компоненты EJB, вызывают метод `create` или `locate` интерфейса `Home` для получения ссылки на слепок класса Remote, который в свою очередь может быть использован для вызова бизнес-методов. То, что метод `create` приводит к созданию ресурсов Enterprise Bean, говорит о реализации в EJB шаблона Builder. С технической точки зрения, этот шаблон ничем не отличается от шаблона Factory Method, поскольку он также обычно используется для создания и сопровождения базовой среды для компонентов Enterprise Bean. Например, вызов метода `create` для компонента Stateful Session Bean приводит к созданию как самого компонента, так и его удаленной реализации.

Proxy (стр. 209). С концептуальной точки зрения, технология Enterprise JavaBeans обеспечивает поведение в стиле шаблона Proxy, поскольку вызовы методов и интерфейса `Home`, и интерфейса Remote, в конечном итоге, транслируются в вызовы нижележащего компонента Enterprise Bean. Конечно, при этом выполняется некоторая дополнительная обработка, а также происходит изменение имени вызываемого метода, поэтому нужно отметить, что полученная структура не в точности представляет классическую реализацию шаблона Proxy.

Session (стр. 231). Компоненты EJB обеспечивают поддержку шаблона Session через компоненты Stateful Session Bean. Необходимо понимать, что этот шаблон применяется к компонентам Session Bean, которые явно были спроектированы для работы в качестве ориентированных на состояние сеансов. Поскольку компоненты Stateless Session Bean не поддерживают концепции единого подхода к идентификации вызывающих объектов, данные такого компонента, даже если они будут сохранены на длительное время, нельзя связать с конкретным клиентом.

Использование шаблона Factory Method коннекторами

J2EE предоставляет в распоряжение разработчика значительное количество API, позволяющих соединить различные технологии масштаба предприятия. Это выглядит вполне естественно, если учесть, что одна из заявленных целей J2EE, заложенных в архитектуре этой редакции языка Java, — обеспечение интеграции как можно большего количества программных систем, работающих на предприятии. Базовая модель для технологий коннекторов проверена временем, поскольку она представляет собой улучшенную модель, которая впервые была применена еще в пакете JDK 1.1, поставляемом с JDBC.

API технологий коннекторов определяет некую программную абстракцию — программный слой, находящийся между приложением Java и некоторой низкоуровневой реализацией. Реализация может быть целой системой или отдельной службой, но чаще всего в качестве реализации используют транслятор — адаптирующий модуль, обеспечивающий взаимодействие API и реального конечного ресурса.

Определив таким образом программную модель, можно создать сравнительно универсальный API, пригодный к использованию во многих программных системах. Получение такого API, обладающего универсальностью, позволяет применить его в нескольких реализациях в рамках одного семейства технологий.

В приведенном ниже списке перечислены API J2EE, представляющие технологии коннекторов.

- Java Messaging Service (JMS). API коннектора для асинхронного обмена сообщениями. В качестве примеров можно привести JMQueue и JMX.
- JavaMail. Коннектор для технологий электронной почты (например, протокол POP3).
- Connector Architecture. Универсальный API коннектора, поддерживающий различные ресурсы Enterprise Information System (EIS) (например, нереляционные базы данных и системы ERP).

Двумя другими технологиями коннекторов J2EE, уже описанными в данной части книги, являются:

- Java Database Connectivity API (JDBC);
- Java Naming and Directory Interface (JNDI).

Некоторые шаблоны проектирования постоянно реализуются на разных уровнях технологий вне зависимости от того, какая именно технология используется. Это объясняется самой природой универсальной архитектуры и распределенной модели, стоящих за моделью коннектора. Поскольку большинство технологий коннекторов для предоставления вызывающим объектам возможности установления начального соединения используют JNDI, коннекторы, как правило, содержат некий механизм установления соединения, который реализует шаблон проектирования Factory Method. Кроме того, учитывая саму концепцию коннектора, нетрудно предположить, что большинство из них в какой-то мере реализуют шаблон Adapter, как минимум, на уровне архитектуры.

Использование шаблонов на уровне архитектуры

Несколько шаблонов реализуются в J2EE скорее на уровне архитектуры системы в целом, чем на уровне каких-то отдельных технологий. Поскольку J2EE работает как федеративная модель предприятия, многие из этих шаблонов можно применить в нескольких или даже во многих местах системы, построенной на базе J2EE. Именно благодаря гибкой архитектурной модели J2EE, состоящей из нескольких объединяемых в одно целое слоев, API, которые поддерживают такие шаблоны, могут использоваться в разных местах системы.

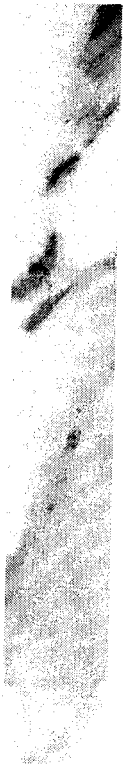
Transaction (стр. 274). Часто коннекторы связывают приложение, выполненное на основе J2EE, с какими-то другими системами, поддерживающими управление транзакциями. Программный интерфейс Java Transaction API (JTA) обеспечивает поддержку координации распределенных транзакций, которую часто называют *двухэтапным подтверждением* (two-phase commit). Технологии, которые поддерживают данный API, должны обеспечивать работы универсальных серверов транзакций. Это, в свою очередь, означает, что они должны поддерживать схему "подтверждение/откат" (Commit/Rollback).

Session (стр. 231). В J2EE также реализуется шаблон Session. Цель реализации этого шаблона в модели J2EE очевидна: хранение промежуточных данных — обычно это очень важная задача в приложениях масштаба предприятия. Для выполнения большинства нетривиальных бизнес-операций требуется сохранение в той или иной форме промежуточных результатов, чтобы сохранить и состояние системы на этапе перехода от одной фазы операции к другой или для гарантии того, что в процессе работы данные не будут потеряны или повреждены.

В типичном приложении масштаба предприятия данные, предназначенные для кратковременного хранения, размещаются на стороне клиента, а данные, для которых должно обеспечиваться долгосрочное хранение, — в базе данных или в других ресурсах EIS. Где-то посредине между этими данными находятся данные, связанные с бизнес-процессами, — иными словами, данные, необходимые для выполнения какой-то работы, которая требует проведения нескольких операций. Именно сохранение промежуточных данных и обеспечивает шаблон Session. Реализация данного шаблона позволяет обеспечить сохранение обрабатываемой информации, которая появляется в результате взаимодействия клиента с системой. Поскольку J2EE подразумевает использование многоуровневой модели приложения, можно обеспечить сохранение данных сеанса на разных уровнях.

- **На уровне клиента.** Клиенты Web J2EE могут использовать cookie-файлы для сохранения информации о сеансах.
- **На уровне Web.** API сервлетов содержит определение интерфейса HttpSession, который позволяет решить задачу сохранения данных клиента.
- **На уровне EJB.** В соответствии со спецификациями EJB, решение этой задачи обеспечивают компоненты Stateful Session Bean.

ПРИМЕРЫ



Приложение A

Системные требования

В данном приложении приведен полный код примеров, использованных в этой книге. В большинстве примеров из первой части книги рассматривались лишь программные фрагменты, необходимые для понимания принципов применения того или иного шаблона. В этом приложении приведены все файлы классов, а также класс RunPattern, необходимый для запуска примера и снабженный многочисленными операторами вывода, позволяющими наглядно продемонстрировать все, что происходит во время работы.

В примерах использования некоторых шаблонов применяется технология RMI (Remote Method Invocation), обеспечивающая возможность удаленного вызова методов. В число таких шаблонов входят: Callback, HOPP, Router, Session, Successive Update, Transaction и Worker Thread.

Для запуска этих примеров необходимо иметь компьютер с настроенным сетевым интерфейсом. В частности, система должна иметь возможность работать с сокетами TCP/IP, а также корректно обрабатывать имя localhost как представление собственного IP-адреса.

Из файла RunPattern во всех примерах каждый раз запускается утилита rmiregistry. Поскольку rmiregistry— это серверный процесс, после завершения примеров будет создаваться впечатление, что система зависла. Для устранения этого эффекта необходимо вручную завершить процесс Java, чтобы снять процесс rmiregistry.

Производящие шаблоны

Abstract Factory

Ниже приведен программный код, на примере которого показано, как с использованием шаблона Abstract Factory можно организовать поддержку иностранных форматов адресов и номеров телефонов в PIM-приложении. Сам механизм создается с помощью интерфейса `AddressFactory` (листинг А.1).

Листинг А.1. AddressFactory.java

```
1. public interface AddressFactory{
2.   public Address createAddress();
3.   public PhoneNumber createPhoneNumber();
4. }
```

Обратите внимание на то, что в интерфейсе AddressFactory определяется два метода, обеспечивающих работу механизма: createAddress и createPhoneNumber. Эти методы генерируют абстрактные продукты Address (листинг А.2) и PhoneNumber (листинг А.3), на которые и возлагается задача определения функциональности самих продуктов.

Листинг А.2. Address.java

```
1. public abstract class Address{
2.   private String street;
3.   private String city;
4.   private String region;
5.   private String postalCode;
6.
7.   public static final String EOL_STRING =
8.      System.getProperty("line.separator");
9.   public static final String SPACE = " ";
10.
11.  public String getStreet(){ return street; }
12.  public String getCity(){ return city; }
13.  public String getPostalCode(){ return postalCode; }
14.  public String getRegion(){ return region; }
15.  public abstract String getCountry();
16.
17.  public String getFullAddress(){
18.     return street + EOL_STRING +
19.       city + SPACE + postalCode + EOL_STRING;
20. }
21.
22.  public void setStreet(String newStreet){ street = newStreet; }
23.  public void setCity(String newCity){ city = newCity; }
24.  public void setRegion(String newRegion){ region = newRegion; }
25.  public void setPostalCode(String newPostalCode){ postalCode =
   newPostalCode; }
26.}
```

Листинг A.3. PhoneNumber.java

```
1. public abstract class PhoneNumber{
2.   private String phoneNumber;
3.   public abstract String getCountryCode();
4.
5.   public String getPhoneNumber(){ return phoneNumber; }
6.
7.   public void setPhoneNumber(String newNumber){
8.     try{
9.        Long.parseLong(newNumber);
10.       phoneNumber = newNumber;
11.    }
12.    catch (NumberFormatException exc){
13.    }
14. }
15.}
```

В рассматриваемом примере Address и PhoneNumber— это абстрактные классы, но в тех ситуациях, когда нет необходимости помещать в них код, предназначенный для работы во всех конкретных продуктах, их можно определить в виде интерфейсов.

Для того чтобы обеспечить в разрабатываемой системе какую-то конкретную функциональность, необходимо создать классы конкретного механизма и конкретного продукта. В нашем примере мы определим класс `USAddressFactory` (листинг A.4), который реализует интерфейс AddressFactory, а также подклассы USAddress (листинг A.5) и `USPhoneNumber` (листинг A.6) классов Address и PhoneNumber, соответственно.

Листинг A.4. USAddressFactory.java

```
1. public class USAddressFactory implements AddressFactory{
2.   public Address createAddress(){
3.      return new USAddress();
4.   }
5.
6.   public PhoneNumber createPhoneNumber(){
7.      return new USPhoneNumber();
8.   }
9. }
```

Листинг A.5. USAddress.java

```
1. public class USAddress extends Address{
2.   private static final String COUNTRY = "UNITED STATES";
3.   private static final String COMMA = ",";
4.
5.   public String getCountry(){ return COUNTRY; }
6.
7.   public String getFullAddress(){
8.      return getStreet() + EOL_STRING +
9.        getCityO + COMMA + SPACE + getRegion() +
10.       SPACE + getPostalCode() + EOL_STRING +
11.       COUNTRY + EOL_STRING;
12. }
13.}
```

Листинг А.6. USPhoneNumber.java

```
1. public class USPhoneNumber extends PhoneNumber{
2.   private static final String COUNTRY_CODE = "01";
3.   private static final int Number_LENGTH = 10;
4.
5.   public String getCountryCode(){ return COUNTRY_CODE; }
6.   public void setPhoneNumber(String newNumber){
7.      if (newNumber.length() == NUMBER_LENGTH){
8.        super.setPhoneNumber) ;
9.      }
10. }
11.}
```

Полученный набор классов AddressFactory, Address и PhoneNumber облегчает процесс расширения возможностей системы для поддержки форматов адреса и номеров телефонов, принятых в других странах. Теперь для того, чтобы добавить функции поддержки форматов какой-либо дополнительной страны, достаточно определить конкретный класс механизма и соответствующий ему конкретный класс продукта. Например, в листингах А.7-А.9 приведены классы, которые можно использовать для добавления поддержки форматов, принятых во Франции.

Листинг А.7. FrenchAddressFactory.java

```
1. public class FrenchAddressFactory implements AddressFactory{
2.   public Address createAddress(){
3.      return new FrenchAddress() ;
4.   }
5.
6.   public PhoneNumber createPhoneNumber(){
7.      return new FrenchPhoneNumber();
8.   }
9. }
```

Листинг А.8. FrenchAddress.java

```
1. public class FrenchAddress extends Address{
2.   private static final String COUNTRY = "FRANCE";
3.
4.   public String getCountry(){ return COUNTRY; }
5.
6.   public String getFullAddress(){
7.      return getStreet() + EOL_STRING +
8.        getPostalCode() + SPACE + getCity() +
9.        EOL_STRING + COUNTRY + EOL_STRING;
10. }
11.}
```

Листинг A.9. FrenchPhoneNumber.java

```
1. public class FrenchPhoneNumber extends PhoneNumber{
2.   private static final String COUNTRY_CODE = "33";
3.   private static final int NUMBER_LENGIH = 9;
4.
5.   public String getCountryCode(){ return COUNTRY_CODE; }
6.
7.   public void setPhoneNumber(String newNumber){
8.      if (newNumber.length() == NUMBER_LENGTH{
9.        super.setPhoneNumber(newNumber);
10.     }
11.  }
12.}
```

Класс RunPattern (листинг A.10) обеспечивает выполнение кода, реализующего рассматриваемый шаблон. Этот класс использует классы USAddressFactory и FrenchAddressFactory для создания двух разных наборов комбинаций адресов и телефонных номеров. Необходимо заметить, что после загрузки производящих объектов можно работать с их продуктами, используя интерфейсы Address и PhoneNumber. Из примера видно, что при обращении к методам классов USAddress и FrenchAddress вызовы этих методов записываются абсолютно одинаково.

Листинг A.10. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.      System.out.println ("Example for the Abstract Factory pattern");
4.      System.out.println ();
5.      System.out.println("  (take a look in the RunPattern code. Notice that
   you can");
6.      System.out.println(" use the Address and PhoneNumber classes when
   writing");
7.      System.out.println(" almost all of the code. This allows you to write a
   very");
8.      System.out.println(" generic framework, and plug in Concrete
   Factories");
9.      System.out.println(" and Products to specialize the behavior of your
   code)");
10.     System.out.println();
11.
12.     System.out.println("Creating U.S. Address and Phone Number:");
13.     AddressFactory usAddressFactory = new USAddressFactory();
14.     Address usAddress = usAddressFactory.createAddress();
15.     PhoneNumber usPhone = usAddressFactory.createPhoneNumber();
16.
17.    usAddress.setStreet("142 Lois Lane");
18.     usAddress.setCity("Metropolis");
19.     usAddress.setRegion("WY");
20.     usAddress.setPostalCode("54321");
21.     usPhone.setPhoneNumber("7039214722");
22.
23.     System.out.println("U.S. address:");
24.    System.out.println(usAddress.getFullAddress() ) ;
25.     System.out.println("U.S. phone number:");
26.    System.out.println(usPhone.getPhoneNumber() ) ;
27.     System.out.println();
28.     System.out.println() ;
```

```
29.
30.   System.out.println("Creating French Address and Phone Number:");
31.   AddressFactory frenchAddressFactory = new FrenchAddressFactory();
32.   Address frenchAddress = frenchAddressFactory.createAddress();
33.   PhoneNumber frenchPhone = frenchAddressFactory.createPhoneNumber() ;
34.
35.   frenchAddress.setStreet("21 Rue Victor Hugo");
36.    frenchAddress.setCity("Courbevoie");
37.    frenchAddress.setPostalCode("40792");
38.    frenchPhone.setPhoneNumber("011324290");
39.
40.   System.out.println("French address:");
41.   System.out.println(frenchAddress.get_FullAddress());
42.   System.out.println("French phone number:");
43.    System.out.println(frenchPhone.getPhoneNumber());
44. }
45.}
```

Builder

В приведенном ниже примере показано, как можно использовать шаблон Builder для создания объекта, представляющего в нашем PIM-приложении некое запланированное пользователем событие. Использованные в примере классы имеют следующее назначение.

- AppointmentBuilder, MeetingBuilder — классы генераторов.
- Scheduler — класс диспетчера.
- Appointment — продукт.
- Address, Contact — вспомогательные классы, используемые для хранения информации, необходимой классу Appointment.
- InformationRequiredException — класс исключительной ситуации Exception, который генерируется в случае нехватки необходимых данных.

Основа шаблона — это класс AppointmentBuilder (листинг А.11), который управляет созданием комплексного продукта, представленном в данном примере объектом Appointment (листинг А. 12). Класс AppointmentBuilder использует серию методов создания отдельных элементов события (buildAppointment, buildLocation, buildDates и buildAttendees), в результате вызова которых формируются объект Appointment с соответствующими данными.

Листинг А. 11. AppointmentBuilder.java

```
1. import java.util.Date;
2. import java.util.ArrayList;
3.
4. public class AppointmentBuilder{
5.
6.   public static final int START_DATE_REQUIRED = 1;
7.   public static final int END_DATE_REQUIRED = 2;
8.   public static final int DESCRIPTION_REQUIRED = 4;
9.   public static final int ATTENDEE_REQUIRED = 8;
10.  public static final int LOCATION_REQUIRED = 16;
```

```
11.
12. protected Appointment appointment;
13.
14. protected int requiredElements;
15.
16. public void buildAppointment(){
17.   appointment = new Appointment();
18. }
19.
20. public void buildDates(Date startDate, Date endDate){
21.   Date currentDate = new Date();
22.   if ((startDate != null) && (startDate.after(currentDate))){
23.     appointment.setStartDate(startDate);
24.   }
25.   if ((endDate != null) && (endDate.after(startDate))){
26.     appointment.setEndDate(endDate);
27.   }
28. }
29.
30. public void buildDescription(String newDescription){
31.   appointment.setDescription(newDescription);
32. }
33.
34. public void buildAttendees(ArrayList attendees) {
35.   if ((attendees != null) &S (!attendees.isEmpty())){
36.     appointment.setAttendees(attendees);
37.   }
38. }
39.
40. public void buildLocation(Location newLocation){
41.   if (newLocation != null){
42.     appointment.setLocation(newLocation);
43.   }
44. }
45.
46. public Appointment getAppointment() InformationRequiredException{
47.   requiredElements = 0;
48.
49.   if (appointment.getStartDate() == null){
50.     requiredElements += START_DATE_REQUIRED;
51.   }
52.
53.   if (appointment.getLocation() == null){
54.     requiredElements += LOCATION_REQUIRED;
55.   }
56.
57.   if (appointment.getAttendees().isEmpty()){
58.     requiredElements += ATTENDEE_REQUIRED;
59.   }
60.
61.   if (requiredElements > 0) {
62.     throw new InformationRequiredException(requiredElements);
63.   }
64.   return appointment;
65. }
66.
67. public int getRequiredElements(){ return requiredElements; }
68. }
```

Листинг А. 12. Appointment.java

```
1. import java.util.ArrayList;
2. import java.util.Date;
3. public class Appointment{
4.   private Date startDate;
5.   private Date endDate;
6.   private String description;
7.   private ArrayList attendees = new ArrayList ();
8.   private Location location;
9.   public static final String EOL_STRING =
10.     System.getProperty("line.separator");
11.
12.  public Date getStartDate(){ return startDate; }
13.  public Date getEndDate(){ return endDate; }
14.  public String getDescription(){ return description; }
15.  public ArrayList getAttendees(){ return attendees; }
16.  public Location getLocation() { return Location; }
17.
18.  public void setDescription(String new Description){ description =
   newDescription; )
19.  public void setLocation(Location newLocation)( location = newLocation; }
20.  public void setStartDate(Date newStartDate){ startDate = newStartDate; }
21.  public void setEndDate(Date newEndDate){ endDate = newEndDate; }
22.  public void setAttendees(ArrayList newAttendees){
23.     if (newAttendees != null){
24.       attendees = newAttendees;
25.     )
26. }
27.
28.  public void addAttendee(Contact attendee){
29.    if (!attendees.contains(attendee) ){
30.      attendees.add(attendee) ;
31.     )
32. >
33.
34.  public void removeAttendee(Contact attendee){
35.    attendees.remove(attendee) ;
36. }
37.
38.  public String toString()(
39.     return " Description: " + description + EOL_STRING +
40.       " Start Date: " + startDate + EOL_STRING +
41.       " End Date: " + endDate + EOL_STRING +
42.       " Location: " + Location + EOL_STRING +
43.       " Attendees: " + attendees;
44. }
45.}
```

Класс Scheduler (листинг А. 13). вызывает методы класса AppointmentBuilder, управляя ходом процесса с помощью метода createAppointment.

Листинг А. 13. Scheduler.java

```
1. import java.util.Date;
2. import java.util.ArrayList;
3. public class Scheduler{
4.   public Appointment createAppointment(AppointmentBuilder builder,
5.     Date startDate, Date endDate, String description,
```

```
6.  Location location, ArrayList attendees) throws
  InformationRequiredException{
7.     if (builder == null){
8.     builder = new AppointmentBuilder();
9.      }
10.    builder.buildAppointment();
11.   builder.buildDates(startDate, endDate);
12  .  builder.buildDescription(description);
13.   builder.buildAttendees(attendees) ;
14.    builder.buildLocation(Location);
15.    return builder.getAppointment();
16. }
17.}
```

Таким образом, классы решают следующие задачи.

- Scheduler. Вызывает нужные методы класса AppointmentBuilder, необходимые для создания объекта Appointment; возвращает ссылку на созданный объект Appointment.
- AppointmentBuilder. Содержит методы создания элементов объекта и применения бизнес-правил; выполняет собственно генерацию объекта Appointment.
- Appointment. Содержит информацию о событии.

Приведенный в листинге А.14 класс MeetingBuilder позволяет увидеть на практике одно из преимуществ шаблона Builder, состоящее в том, что для того, чтобы добавить дополнительные правила к классу Appointment, достаточно расширить уже имеющийся генератор. В данном примере класс MeetingBuilder накладывает на планируемое событие дополнительные ограничения, а именно — при планировании такого события, как деловая встреча, должны обязательно указываться данные о времени начала и конца этого события.

Листинг А.14. MeetingBuilder.java

```
1. import java.util.Date;
2. import java.util.Vector;
3.
4. public class MeetingBuilder extends AppointmentBuilder{
5.   public Appointment getAppointment() throws InformationRequiredException{
6.     try{
7.        super.getAppointment();
8.      }
9.      finally{
10.      if (appointment.getEndDate() == null){
11.           requiredElements += END_DATE_REQUIRED;
12.       )
13.
14.     if (requiredElements > 0) {
15.          throw new InformationRequiredException(requiredElements);
16.       }
17.    }
18.    return appointment;
19. }
20.}
```

Ко вспомогательным классам, используемым в данном примере, относится InformationRequiredException (листинг А.15), а также интерфейсы Location (листинг А. 16) и Contact (листинг А.18). Два последних интерфейса в данном примере играют вспомогательную роль, представляя информацию, необходимую классу Appointment. Реализация этих интерфейсов представлена классами Locationlmpl (листинг А.17) и Contactlmpl (листинг А.18).

Листинг А.15. InformationRequiredException.java

```
1. public class InformationRequiredException extends Exception{
2.   private static final String MESSAGE = "Appointment cannot be created
   because further information is required";
3.   public static final int START_DATE_REQUIRED = 1;
4.   public static final int END_DATE_REQUIRED = 2;
5.   public static final int DESCRIPTION_REQUIRED = 4;
6.   public static final int ATTENDEE_REQUIRED = 8;
7.   public static final int LOCATION_REQUIRED = 16;
8.   private int informationRequired;
9.
10.  public InformationRequiredException(int itemsRequired){
11.     super(MESSAGE);
12.     informationRequired = itemsRequired;
13. }
14.
15.  public int getInformationRequired(){ return informationRequired; }
16.}
```

Листинг А. 16. Location.java

```
1. import java.io.Serializable;
2. public interface Location extends Serializable{
3.   public String getLocation();
4.   public void setLocation(String newLocation);
5. }
```

Листинг А. 17. LocationImpl.java

```
1. public class LocationImpl implements Location;
2.   private String location;
3.
4.   public LocationImpl(){ }
5.   public LocationImpl(String newLocation){
6.      location = newLocation;
7. }
8.
9.   public String getLocation(){ return location; }
10.
11.  public void setLocation(String newLocation){ location = newLocation; }
12.
13.  public String toString(){ return location; }
14.}
```

Листинг A.18. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization) ;
13.}
```

Листинг A.19. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(String newFirstName, String newLastName,
8.      String newTitle, String newOrganization){
9.        firstName = newFirstName;
10.       lastName = newLastName;
11.       title = newTitle;
12.       organization = newOrganization;
13.  }
14.
15.  public String getFirstName(){ return firstName; }
16.  public String getLastName(){ return lastName; }
17.  public String getTitle(){ return title; }
18.  public String getOrganization(){ return organization; }
19.
20.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
21.  public void setLastName(String newLastName){ lastName = newLastName; }
22.  public void setTitle(String newTitle){ title = newTitle; }
23.  public void setOrganization(String newOrganization){ organization =
   newOrganization;}
24.
25.  public String toString(){
26.     return firstName + SPACE + lastName;
27.  }
28.}
```

Файл RunPattern (листинг A.20) запускает данный пример на выполнение. В примере показано, как с помощью шаблона Builder можно создать три разных объекта класса Appointment, используя классы AppointmentBuilder и MeetingBuilder.

Листинг А.20. RunPattern.java

```
1. import java.util.Calendar;
2. import java.util.Date;
3. import java.util.ArrayList;
4. public class RunPattern{
5.   private static Calendar dateCreator = Calendar.getInstance();
6.
7.   public static void main(String [] arguments){
8.     Appointment appt = null;
9.
10.    System.out.println("Example for the Builder pattern");
11.    System.out.println();
12.    System.out.println("This example demonstrates the use of the Builder");
13.    System.out.println("pattern to create Appointment objects for the PIM.");
14.    System.out.println ();
15.
16.    System.out.println("Creating a Scheduler for the example.");
17.    Scheduler pimScheduler = new Scheduler!);
18.
19.    System.out.println("Creating an AppointmentBuilder for the example.");
20.    System.out.println();
21.    AppointmentBuilder apptBuilder = new AppointmentBuilder();
22.    try{
23.      System.out.println("Creating a new Appointment with an
   AppointmentBuilder");
24.      appt = pimScheduler.createAppointment(
25.         apptBuilder, createDate(2066, 9, 22, 12, 30),
26.         null, "Trek convention", new LocationImpl("Fargo, ND"),
27.           createAttendees(4));
28.      System.out.println("Successfully created an Appointment.");
29.      System.out.println("Appointment information:");
30.      System.out.println(appt) ;
31.      System.out.println();
32.    }
33.    catch (InformationRequiredException exc) {
34.      printExceptions(exc) ;
35.}
36.
37.    System.out.println("Creating a MeetingBuilder for the example.");
38.    MeetingBuilder mtgBuilder = new MeetingBuilder();
39.    try{
40.      System.out.println("Creating a new Appointment with a
   MeetingBuilder");
41.      System.out.println("(notice that the same create arguments will
   produce");
42.     System.out.println(" an exception, since the MeetingBuilder enforces a");
43.      System.out.println(" mandatory end date)");
44.      appt = pimScheduler.createAppointment(
45.         mtgBuilder, createDate(2066, 9, 22, 12, 30),
46.         null, "Trek convention", new LocationImpl("Fargo, ND"),
47.           createAttendees(4));
48.      System.out.println("Successfully created an Appointment.");
49.      System.out.println("Appointment information:");
50.      System.out.println(appt);
51.      System.out.println() ;
52.    }
53.    catch (InformationRequiredException exc)(
54.      printExceptions(exc) ;
55.    }
56.
57.    System.out.println("Creating a new Appointment with a MeetingBuilder");
```

```
58.   System.out.println("(This time, the MeetingBuilder will provide an end
   date)");
59.    try{
60.     appt = pimScheduler.createAppointment(
61.          mtgBuilder,
62.        createDate(2002, 4, 1, 10, 00),
63.        createDate(2002, 4, 1, 11, 30),
64.          "000 Meeting",
65.         new LocationImpl("Butte, MT"),
66.         createAttendees(2) );
67.     System.out.println("Successfully created an Appointment.");
68.      System.out.println("Appointment information:");
69.      System.out.println(appt);
70.      System.out.println();
71.    }
72.    catch (InformationRequiredException exc){
73.     printExceptions(exc);
74.    }
75. }
76.
77. public static Date createDate(int year, int month, int day, int hour, int
   minute){
78.   dateCreator.set(year, month, day, hour, minute);
79.   return dateCreator.getTime() ;
80. }
81.
82. public static ArrayList createAttendees(int numberToCreate))
83.   ArrayList group = new ArrayList();
84.   for (int i = 0; i < numberToCreate; i++){
85.    group.add(new ContactImpl("John", getLastName(i), "Employee
   (nonexempt)", "Yoyodyne Corporation"));
86.    }
87.   return group;
88. }
89.
90. public static String getLastName(int index){
91.   String name = "";
92.   switch (index % 6){
93.    case 0: name = "Worfin";
94.          break;
95.    case 1: name = "Smallberries";
96.          break;
97.    case 2: name = "Bigbootee";
98.          break;
99.    case 3: name = "Haugland";
100.         break;
101.   case 4: name = "Maassen";
102.         break;
103.   case 5: name = "Sterling";
104.         break;
105.  }
106.  return name;
107. }
108.
109. public static void printExceptions(InformationRequiredException exc){
110.   int statusCode = exc.getInformationRequired();
111.
112.   System.out.println ("Unable to create Appointment: additional information
   is required");
113.    if ((statusCode & InformationRequiredException.START_DATE_REQUIRED) > 0){
114.      System.out.println(" A start date is required for this
   appointment to be complete.");
115.    }
```

```
116.    if ( (statusCode & InformationRequiredException.END_DATE_REQUIRED) > 0){
117.        System.out.println(" An end date is required for this appointment
    to be complete.");
118.    }
119.    if ((statusCode S InformationRequiredException.DESCRIPTION_REQUIRED) > 0){
120.        System.out.println(" A description is required for this
    appointment to be complete.");
121.    }
122.    if ((statusCode & InformationRequiredException.ATTENDEE_REQUIRED) > 0){
123.        System.out.println (" At least one attendee is required for this
    appointment to be complete.");
124.    }
125.    if ((statusCode & InformationRequiredException.LOCATION_REQUIRED) > 0){
126.        System.out.println (" A location is required for this appointment
    to be complete.");
127.    }
128.    System.out.println ();
129. }
130. }
```

Factory Method

В приведенном ниже примере показано, как для обеспечения возможности редактирования элемента **PIM-приложения** используется шаблон Factory Method. Такая возможность, очевидно, весьма пригодится пользователю PIM-приложения, которое, по определению, должно управлять самой разной информацией. Для повышения гибкости системы в рассматриваемом приложении используются интерфейсы.

В интерфейсе Editable (листинг А.21) объявлен метод getEditor, возвращающий интерфейс ItemEditor. Достоинством использования интерфейса Editable является то, что с его помощью любой элемент PIM-приложения может предоставить пользователю редактор для изменения своих данных. Это обеспечивается путем генерации объекта, которому известно, какие данные прикладного объекта можно изменять и какие значения являются допустимыми. Единственная задача, которая возлагается на подсистему пользовательского интерфейса— это использование интерфейса Editable для получения редактора.

Листинг А.21. Editable.java

```
1. public interface Editable (
2.   public ItemEditor getEditor ();
3. }
```

В интерфейсе ItemEditor (листинг А.22), в свою очередь, объявлены два метода: getGUI и commitChanges. Метод getGUI – это еще один метод, "фабрикующий" объекты. В данном случае он генерирует объекты класса JComponent, которые представляют собой графический пользовательский интерфейс Swing и обеспечивают редактирование текущего элемента. Это позволяет получить очень гибкую систему: при добавлении элемента нового типа графический пользовательский интерфейс может оставаться неизменным, так как он использует лишь интерфейсы Editable и ItemEditor.

Объект JComponent, возвращаемый методом getGUI, может обладать всеми средствами, необходимыми для редактирования выбранного элемента PIM-приложения.

Подсистеме пользовательского интерфейса остается лишь отобразить полученный объект JComponent, чтобы обеспечить пользователю возможность применения функциональности JComponent для редактирования элемента. Попутно заметим, что так как далеко не каждое приложение работает в графическом режиме, было бы неплохо предусмотреть и наличие альтернативного метода, такого как getUI, возвращающего объект класса Object или другого класса, обеспечивающего пользователю возможность работы с данными выбранного элемента без графического интерфейса.

Второй метод, commitChanges, позволяет пользовательскому интерфейсу сообщить редактору о том, что пользователь решил завершить внесение изменений.

Листинг A.22. ItemEditor.java

```
1. import javax.swing.JComponent;
2. public interface ItemEditor {
3.   public JComponent getGUIO;
4.   public void commitChanges();
5. }
```

В листинге A.23 показан пример возможной реализации одного из элементов *PIM-приложения* — Contact. Класс Contact имеет два атрибута: имя контактного лица и указание на то, кем оно доводится пользователю. На примере этих атрибутов показано, как можно вводить информацию, описывающую элемент PIM-приложения.

Листинг A.23. Contact.java

```
1. import java.awt.GridLayout;
2. import java.io.Serializable;
3. import javax.swing.JComponent;
4. import javax.swing.JLabel;
5. import javax.swing.JPanel;
6. import javax.swing.JTextField;
7.
8. public class Contact implements Editable, Serializable {
9.   private String name;
10.  private String relationship;
11.
12.  public ItemEditor getEditor() {
13.     return new ContactEditor();
14.  )
15.
16.  private class ContactEditor implements ItemEditor, Serializable (
17.     private transient JPanel panel;
18.     private transient JTextField nameField;
19.     private transient JTextField relationField;
20.
21.     public JComponent getGUI() {
22.       if (panel == null) {
23.            panel = new JPanel();
24.            nameField = new JTextField(name);
25.            relationField = new JTextField(relationship);
26.            panel.setLayout(new GridLayout(2,2));
27.            panel.add(new JLabel("Name:"));
28.            panel.add(nameField);
29.            panel.add(new JLabel("Relationship:"));
30.           panel.add(relationField) ;
31.       } else {
```

```
32.           nameField.setText(name);
33.           relationField.setText(relationship);
34.       }
35.       return panel;
36.     }
37.
38.     public void commitChanges()  {
39.       if (panel != null)  {
40.           name = nameField.getText();
41.           relationship = relationField.getText();
42.       }
43.     }
44.
45.     public String toString(){
46.       return "\nContact:\n" +
47.           "    Name: " + name + "\n" +
48.           "    Relationship: " + relationship;
49.     }
50. }
51.}
```

Класс Contact реализует интерфейс Editable и предоставляет собственный редактор. Этот редактор может работать лишь с объектами класса Contact, так как он предназначен для изменения определенных атрибутов этого класса. В этой связи он реализован в виде вложенного класса. Такое решение позволяет ему получать доступ к атрибутам внешнего класса. В случае реализации редактора в виде отдельного (т.е. не вложенного) класса, пришлось бы обеспечить класс Contact дополнительными методами, с помощью которых редактор мог бы получать доступ к атрибутам и изменять их значения. Понятно, что такое решение некорректно с точки зрения ограничения доступа к внутренним данным класса Contact.

Обратите внимание на то, что редактор сам по себе не является компонентом Swing, а выполнен в виде объекта, который обеспечивает механизм получения такого компонента. Преимущество такого подхода состоит в том, что разработчик может применять для этого объекта сериализацию и направлять его в поток. Для этого необходимо объявить все Swing-атрибуты класса ContactEditor транзитными (transient), и они будут создаваться в тех случаях, когда в этом возникнет необходимость.

Класс EditorGui (листинг А.24) представляет собой редактор общего назначения, который можно применить в PIM-приложении. Необходимо отметить, что этот класс для управления окном редактирования использует интерфейс ItemEditor. Он создает экземпляр класса JPanel для организации окна редактирования и помещает на него экземпляр класса JComponent, полученный с помощью вызова метода getGUI. Все средства редактирования экземпляра класса Contact обеспечиваются Swing-компонентом, а класс EditorGui предоставляет лишь управляющие кнопки и область JTextArea, в которой отображается состояние объекта Contact.

Листинг А.24. EditorGui.java

```
1. import java.awt.Container;
2. import java.awt.event.ActionListener;
3. import java.awt.event.WindowAdapter;
4. import java.awt.event.ActionEvent;
5. import java.awt.event.WindowEvent;
6. import javax.swing.BoxLayout;
```

```
7. import javax.swing.JButton;
8. import javax.swing.JComponent;
9. import javax.swing.JFrame;
10.import javax.swing.JPanel;
11.import javax.swing.JTextArea;
12.public class EditorGui implements ActionListener{
13. private JFrame mainFrame;
14. private JTextArea display;
15. private JButton update, exit;
16. private JPanel controlPanel, displayPanel, editorPanel;
17. private ItemEditor editor;
18.
19. public EditorGui(ItemEditor edit){
20.   editor = edit;
21. }
22.
23. public void createGui(){
24.   mainFrame = new JFrame("Factory Pattern Example");
25.   Container content = mainFrame.getContentPane();
26.   content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
27.
28.   editorPanel = new JPanel();
29.  editorPanel.add(editor.getGUI() ) ;
30.   content.add(editorPanel) ;
31.
32.   displayPanel = new JPanel();
33.   display = new JTextArea(10, 40);
34.    display.setEditable(false);
35.    displayPanel.add(display);
36.   content.add(displayPanel) ;
37.
38.    controlPanel = new JPanel();
39.   update = new JButton("Update Item");
40.   exit = new JButton("Exit");
41.    controlPanel.add(update);
42.    controlPanel.add(exit);
43.   content.add(controlPanel) ;
44.
45.   update.addActionListener(this) ;
46.   exit.addActionListener(this) ;
47.
48.    mainFrame.addWindowListener(new WindowCloseManager());
49.    mainFrame.pack();
50.    mainFrame.setVisible(true);
51. }
52.
53.
54. public void actionPerformed(ActionEvent evt){
55.   Object originator = evt.getSource();
56.   if (originator == update){
57.     updateItem() ;
58.    }
59.   else if (originator == exit){
60.     exitApplication();
61.    i
62. }
63.
64. private class WindowCloseManager extends WindowAdapter{
65.   public void windowClosing(WindowEvent evt){
66.     exitApplication();
67.    }
68. }
69.
```

```
70. private void updateItem(){
71.    editor.commitChanges();
72.    display.setText(editor.toString());
73. }
74.
75. private void exitApplication(){
76.    System.exit(0);
77. }
78.}
```

Обратите внимание на то, что нажатие кнопки **Update Item** приводит к вызову метода commitChanges редактора.

Класс RunPattern (листинг А.25) запускает данный пример на выполнение и создает объекты Contact и EditorGui. Конструктор последнего создает в рассматриваемом примере экземпляр класса ItemEditor.

Листинг А.25. RunPattern.java

```
1. import javax.swing.JComponent;
2. import javax.swing.JFrame;
3. import java.awt.event.WindowAdapter;
4. import java.awt.event.WindowEvent;
5.
6.
7. public class RunPattern{
8.  public static void main(String [] arguments){
9.     System.out.println("Example for the FactoryMethod pattern");
10.    System.out.println();
11.
12.    System.out.println("Creating a Contact object");
13.    System.out.println();
14.    Contact someone = new Contact();
15.
16.    System.out.println("Creating a GUI editor for the Contact");
17.    System.out.println();
18.    System.out.println("The GUI defined in the EditorGui class is a truly
   generic editor.");
19.    System.out.println("It accepts an argument of type ItemEditor, and
   delegates");
20.    System.out.println(" all editing tasks to its ItemEditor and the
   associated GUI.");
21.    System.out.println (" The getEditor() Factory Method is used to obtain
   the ItemEditor");
22.    System.out.println(" for the example.");
23.    System.out.println();
24.    System.out.println("Notice that the editor in the top portion of the GUI
   is,");
25.    System.out.println(" in fact, returned by the ItemEditor belonging to
   the");
26.    System.out.println(" Contact class, and has appropriate fields for that
   class.");
27.
28.    EditorGui runner = new EditorGui(someone.getEditor());
29.    runner.createGui();
30. }
31.}
```

Prototype

Примером использования шаблона Prototype в данном примере является класс Address, который позволяет создавать новую адресную запись на основе имеющейся. Базовая функциональность шаблона определяется интерфейсом CopyaЫe (листинг А.26).

Листинг А.26. Copyable.java

```
1. public interface Copyable{
2.   public Object copyO;
3. }
```

Интерфейс CopyaЫe содержит определение метода copy и обеспечивает определение операции копирования для любого класса, который реализует этот интерфейс. В данном примере мы рассмотрим частичное копирование, т.е. копирование, при котором объектные ссылки оригинального объекта просто дублируются в его копии.

В листинге А.27 продемонстрировано важное свойство операции копирования, состоящее в том, что далеко не во всех ситуациях обязательно копировать все поля без исключения. В данном случае в новый объект не копируется информация о типе адреса, так как пользователь, скорее всего, захочет его изменить, применяя графический интерфейс PIM-приложения.

Листинг А.27. Address.java

```
1. public class Address implements Copyable{
2.   private String type;
3.   private String street;
4.   private String city;
5.   private String state;
6.   private String zipCode;
7.   public static final String EOL_STRING =
8.     System.getProperty("line.separator") ;
9.   public static final String COMMA = ",";
10.  public static final String HOME = "home";
11.  public static final String WORK = "work";
12.
13.  public Address(String initType, String initStreet,
14.     String initCity, String initState, String initZip){
15.       type = initType;
16.       street = initStreet;
17.       city = initCity;
18.       state = initstate;
19.       zipCode = initZip;
20.  }
21.
22.  public Address(String initStreet, String initCity,
23.     String initstate, String initZip){
24.      this(WORK, initStreet, initCity, initstate, initZip);
25.  }
26.  public Address(String initType){
27.     type = initType;
28.  }
29.  public Address (){ }
30.
```

```
31. public String getType(){ return type; }
32. public String getStreet(){ return street; }
33. public String getCity(){ return city; }
34. public String getState(){ return state; }
35. public String getZipCode(){ return zipCode; }
36.
37. public void setType(String newType){ type = newType; }
38. public void setStreet(String newStreet){ street = newStreet; }
39. public void setCity(String newCity){ city = newCity; }
40. public void setState(String newState){ state = newState; }
41. public void setZipCode(String newZip){ zipCode = newZip; }
42.
43. public Object copy(){
44.   return new Address(street, city, state, zipCode);
45. }
46.
47. public String toString(){
48.   return "\t" + street + COMMA + " " + EOL_STRING +
49.     "\t" + city + COMMA + " " + state + " " + zipCode;
50. }
51.}
```

Пример использования на практике данного шаблона приведен в листинге А.28, который содержит класс RunPattern. Этот класс сначала создает экземпляр класса Address, а затем дублирует его, вызывая метод copy полученного объекта. Тот факт, что экземпляры класса Address возвращают два разных значения хеш-кода (числовые значения, показывающие уникальность сущности объекта), демонстрирует, что в результате операции копирования был получен совершенно новый объект, отличный от первого.

Листинг А.28. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.     System.out.println("Example for Prototype pattern");
4.     System.out.println();
5.     System.out.println("This example will create an Address object,");
6.     System.out.println(" which it will then duplicate by calling the");
7.     System.out.println(" object's clone method.");
8.     System.out.println ();
9.
10.    System.out.println("Creating first address.");
11.    Address address1 = new Address("8445 Silverado Trail", "Rutherford",
   "CA", "91734");
12.    System.out.println("First address created.");
13.    System.out.println(" Hash code = " + address1.hashCode());
14.    System.out.println(address1) ;
15.    System.out.println();
16.
17.    System.out.println("Creating second address using the clone() method.");
18.    Address address2 = (Address)address1.copy();
19.    System.out.println("Second address created.");
20.    System.out.println(" Hash code = " + address2.hashCode());
21.    System.out.println(address2);
22.    System.out.println();
23.
24.
25.  }
26.}
```

Singleton

Пользователям приложения, без сомнения, понадобится функция отмены предыдущих команд. Для того чтобы обеспечить поддержку в приложении этой функциональности, необходимо добавить в него хронологический список операций. Соответствующий объект должен быть доступен из любого другого объекта PIM-приложения, причем все объекты должны работать с одним и тем же единственным экземпляром класса. Таким образом, хронологический список— это практически идеальный кандидат на применение шаблона Singleton (листинг А.29).

Листинг А.29. Singleton.java

```
1. import java.util.ArrayList;
2. import java.util.Collections;
3. import java.util.List;
4. public class HistoryList{
5.   private List history = Collections.synchronizedList(new ArrayList ());
6.   private static HistoryList instance = new HistoryList();
7.
8.   private HistoryList(){ }
9.
10.  public static HistoryList get Instance(){
11.     return instance;
12.  }
13.
14.  public void addCommand(String command){
15.    history.add(command) ;
16.  }
17.
18.  public Object undoCommand(){
19.    return history.remove(history.size () - 1) ;
20.  }
21.
22.  public String toString(){
23.     StringBuffer result = new StringBuffer();
24.     for (int i = 0 ; i < history.size(); i++){
25.      result.append(" " ) ;
26.       result.append(history.get(i)) ;
27.        result.append("\n");
28.     }
29.     return result.toString();
30.  }
31.}
```

Класс HistoryList содержит статическую переменную, в которой хранится ссылка на собственный экземпляр класса, а также закрытый конструктор и метод getInstance, предоставляющий ссылку на единственный объект хронологического списка всем элементам приложения. Еще одна переменная, history, представляет собой объект класса List, который используется для отслеживания текстового представления команд. Кроме того, у класса HistoryList имеется два метода (addCommand и undoCommand), предназначенные соответственно, для добавления команд к списку и удаления их из списка.

Базовый Swing-ориентированный графический пользовательский интерфейс в примере обеспечивает класс SingletonGui (листинг А.30). Данный графический ин-

терфейс поддерживает базовый набор команд: создание контакта, планирование мероприятия, отмена, обновление и выход. При работе с командой создания вызывается статический метод `getInstance` класса `HistoryList`, а затем у полученного объекта вызывается метод `addCommand`. При работе с командой отмены выполняется такая же последовательность операций, с той лишь разницей, что последним вызывается метод `undoCommand`. Метод `refresh`вызывает метод `toString` экземпляра класса `HistoryList` с тем, чтобы получить содержимое хронологического списка для его отображения на экране.

Листинг А.30. `SingletonGui.java`

```
1. import java.awt.Container;
2. import javax.swing.BoxLayout;
3. import javax.swing.JButton;
4. import javax.swing.JFrame;
5. import javax.swing.JPanel;
6. import javax.swing.JTextArea;
7. import java.awt.event.ActionEvent;
8. import java.awt.event.ActionListener;
9. import java.awt.event.WindowAdapter;
10.import java.awt.event.WindowEvent;
11.public class SingletonGui implements ActionListener{
12.  private JFrame mainFrame;
13.  private JTextArea display;
14.  private JButton newContact, newAppointment, undo, refresh, exit;
15.  private JPanel controlPanel, displayPanel;
16.  private static int historyCount;
17.
18.  public void createGui(){
19.    mainFrame = new JFrame("Singleton Pattern Example");
20.    Container content = mainFrame.getContentPane();
21.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
22.
23.    displayPanel = new JPanel();
24.    display = new JTextArea(20, 60);
25.    display.setEditable(false) ;
26.    displayPanel.add(display) ;
27.    content.add(displayPanel) ;
28.
29.    controlPanel = new JPanel ();
30.    newContact = new JButton("Create contact");
31.    newAppointment = new JButton("Create appointment");
32.    undo = new JButton("Undo") ;
33.    refresh = new JButton("Refresh") ;
34.    exit = new JButton("Exit") ;
35.    controlPanel.add(newContact) ;
36.    controlPanel.add(newAppointment) ;
37.    controlPanel.add(undo) ;
38.    controlPanel.add(refresh) ;
39.    controlPanel.add(exit) ;
40.    content.add(controlPanel) ;
41.
42.    newContact.addActionListener(this) ;
43.    newAppointment.addActionListener(this) ;
44.    undo.addActionListener(this) ;
45.    refresh.addActionListener(this) ;
46.    exit.addActionListener(this) ;
47.
48.    mainFrame.addWindowListener(new WindowCloseManager());
```

```
49.    mainFrame.pack();
50.   mainFrame.setVisible(true) ;
51. }
52.
53. public void refreshDisplay(String actionMessage){
54.    display.setText(actionMessage + "\nCOMMAND HISTORY:\n" +
55.     HistoryList.getInstance().toString() ) ;
56. >
57.
58. public void actionPerformed(ActionEvent evt){
59.    Object originator = evt.getSource();
60.    if (originator == newContact){
61.     addCommand(" New Contact");
62.    }
63.    else if (originator == newAppointment){
64.     addCommand(" New Appointment");
65.    }
66.    else if (originator == undo){
67.     undoCommand() ;
68.    }
69.    else if (originator == refresh){
70.      refreshDisplay("");
71. }
72.    else if (originator == exit){
73.      exitApplication();
74.    }
75. }
76.
77. private class WindowCloseManager extends WindowAdapter{
78.    public void windowClosing(WindowEvent evt){
79.      exitApplication();
80.    }
81. }
82.
83. private void addCommand(String message){
84.   HistoryList.getInstance().addCommand((++historyCount)  + message);
85.   refreshDisplay("Add Command: " + message);
86. }
87.
88. private void undoCommand(){
89.     Object result = HistoryList.getInstance().undoCommand();
90.     historyCount--;
91.     refreshDisplay("Undo Command: " + result);
92.  )
93.
94.  private void exitApplication(){
95.     System.exit(0);
96.  }
97.}
```

Запуск примера на выполнение осуществляется с помощью класса RunPattern (листинг А.31). Этот класс создает два экземпляра класса SingletonGui, каждый из которых позволяет работать с одним и тем же хронологическим списком.

Листинг А.31. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.      System.out.println("Example for Singleton pattern");
A.      System.out.println();
5.     System.out.println("This example will demonstrate the use of");
6.     System.out.println(" the Singleton pattern by creating two GUI");
7.     System.out.println(" editors, both of which will reference the");
8.     System.out.println(" same underlying history list.");
9.
10.    System.out.println("Creating the first editor");
11.    System.out.println();
12.    SingletonGui editorl = new SingletonGui();
13.    editor1.createGui();
14.
15.    System.out.println("Creating the second editor");
16.    System.out.println();
17.    SingletonGui editor2 = new SingletonGui();
18.    editor2.createGui() ;
19. }
20.}
```

Поведенческие шаблоны

Chain of Responsibility

PIM-приложение может управлять не только контактами, но и проектами. В данном примере показано, как использовать шаблон Chain of Responsibility для получения информации, хранящейся в иерархической структуре проекта.

Интерфейс `ProjectItem` (листинг А.32) определяет общие методы для любого элемента, который может быть частью проекта.

Листинг А.32. `ProjectItem.java`

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public interface ProjectItem extends Serializable{
4.   public static final String EOL_STRING =
   System.getProperty("line.separator");
5.    public ProjectItem getParent();
6.   public Contract getOwner();
7.   public String getDetails();
8.   public ArrayList getProjectItems();
9. }
```

Интерфейс определяет методы getParent, getOwner, getDetails и getPrcjectItems. В рассматриваемом примере интерфейс ProjectItem реализуется двумя классами— Project и Task. Класс Project (листинг А.33) является базой проекта, поэтому его метод getParent возвращает пустой указатель (null). Методы getOwner и getDetails возвращают, соответственно, владельца и описание всего проекта в целом, а метод `getProjectItems`— ссылки на все элементы проекта нижележащего уровня.

Листинг А.33. `Project.java`

```
1. import java.util.ArrayList;
2. public class Project implements ProjectItem{
3.   private String name;
4.   private Contact owner;
5.   private String details;
6.   private ArrayList projectItems = new ArrayList();
7.
8.  public Project () ()
9.   public Project (String newName, String newDetails, Contact newOwner){
10.     name = newName;
11.     Owner = newOwner;
12.     details = newDetails;
13. }
14.
15.  public String getName(){ return name; }
16.  public String getDetails (){ return details; }
17.  public Contact getOwner(){ return owner; }
18.  public ProjectItem getParent (){ return null; }
19.  public ArrayList getProjectItems(){ return projectItems; }
20.
21.  public void setName(String newName){ name = newName; }
22.  public void setOwner(Contact newOwner){ owner = newOwner; }
23.  public void setDetails(String newDetails){ details = newDetails; }
24.
```

```
25. public void addProjectItem(ProjectItem element){
26.   if (!projectItems.contains(element)){
27.     projectItems.add(element);
28.   }
29. }
30.
31. public void removeProjectItem(ProjectItem element){
32.   projectItems.remove(element);
33. }
34.
35. public String toString(){
36.   return name;
37. }
38.}
```

Класс Task (листинг А.34) представляет собой некоторую отдельную задачу в рамках проекта. Подобно классу `Project`, класс `Task` также хранит коллекцию подзадач, ссылку на которую возвращает его метод `getProjectItems`. Метод `getParent` класса `Task` возвращает ссылка на родительский объект, который может быть экземпляром класса Task или `Project`.

Листинг А.34. `Task.java`

```
1. import java.util.ArrayList;
2. import java.util.ListIterator;
3. public class Task implements ProjectItem{
4.   private String name;
5.   private ArrayList projectItems = new ArrayListO;
6.   private Contact owner;
7.   private String details;
8.   private ProjectItem parent;
9.   private boolean primaryTask;
10.
11.  public Task(ProjectItem newParent){
12.    this (newParent, "", "",null, false);
13.  }
14.  public Task(ProjectItem newParent, String newName,
15.    String newDetails, Contact newOwner, boolean newPrimaryTask){
16.      parent = newParent;
17.      name = newName;
18.      owner = newOwner;
19.      details = newDetails;
20.      primaryTask = newPrimaryTask;
21.  }
22.
23.  public Contact getOwner(){
24.    if (owner == null){
25.      return parent.getOwner()
26.    }
27.    else{
28.      return owner;
29.    }
30.  }
31.
32.  public String getDetails (){
33.    if (primaryTask) {
34.      return details;
35.    }
36.    else{
```

```
37.     return parent.getDetails()  + EOL_STRING + "\t" + details;
38.     }
39. }
40.
41. public String getName(){ return name; }
42. public ArrayList getProjectItems(){ return projectItems; }
43. public ProjectItem getParent(){ return parent;}
44. public boolean isPrimaryTask(){ return primaryTask; }
45.
46. public void setName(String newName){ name = newName; }
47. public void setOwner(Contact newOwner){ owner = newOwner; }
48. public void setParent(ProjectItem newParent){ parent = newParent; }
49. public void setPrimaryTask(boolean newPrimaryTask){ primaryTask =
    newPrimaryTask; }
50. public void setDetails(String newDetails){ details = newDetails; }
51.
52. public void addProjectItem(ProjectItem element){
53.     if (!projectItems.contains(element)){
54.       projectItems.add(element);
55.     }
56. }
57.
58.   public void removeProjectItem(ProjectItem element)}
59.     projectItems.remove(element);
60.   }
61.
62.   public String toString(){
63.     return name;
64.   }
65.}
```

Поведение, характерное для шаблона Chain of Responsibility, проявляется в методах getOwner и getDetails класса Task. Так, метод `getOwner` этого класса возвращает либо значение атрибута владельца своего класса (если оно отлично от нуля), либо значение, предоставленное родительским классом. Если родительским классом является класс `Task` и его атрибут владельца также не установлен, вызов метода передается следующему родителю до тех пор, пока не будет найдено отличное от нуля значение или пока не будет вызван аналогичный метод класса `Project`. Это позволяет без особого труда создать группу задач, представленных экземплярами класса `Task`, владельцем которых будет назначено одно лицо, отвечающее как за исполнение главной задачи Task, так и всех ее подзадач.

Метод `getDetails` также является примером поведения, типичного для шаблона Chain of Responsibility, хотя оно проявляется несколько иначе. Данный метод вызывает метод `getDetails` каждого из родителей до тех пор, пока не достигнет класса Task или `Project`, идентифицированного как терминальный узел. Это означает, что метод `getDetails` возвращает набор объектов класса `String`, совокупность которых представляет описание конкретной задачи с глубиной детализации, соответствующей точке вызова `getDetails`.

Ко вспомогательным классам данного примера относятся интерфейс `Contact` (листинг А.35) и класс `ContactImpl` (листинг А.36), который используется классами `Project` и Task для определения владельца.

Листинг A.35. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг A.36. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14.  }
15.
16.  public String getFirstName(){ return firstName; }
17.  public String getLastName(){ return lastName; }
18.  public String getTitle(){ return title; }
19.  public String getOrganization(){ return organization; }
20.
21.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
22.  public void setLastName(String newLastName){ lastName = newLastName; }
23.  public void setTitle(String newTitle){ title = newTitle; }
24.  public void setOrganization(String newOrganization){ organization =
     newOrganization;}
25.
26.  public String toString(){
27.    return firstName + SPACE + lastName;
28.  }
29.}
```

Класс DataCreator (листинг A.37) содержит ряд вспомогательных классов, предназначенных для генерации данных и сериализации их в файл, а класс DataRetriever (листинг A.38) обеспечивает решение обратной задачи— извлечения данных для использования их в рассматриваемом примере. Класс `RunPattern` (листинг A.39) координирует работу остальных классов: извлекает проект из файла, затем получает владельца и детали по каждой задаче и всему проекту в целом.

Листинг А.37. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5.
6. public class DataCreator{
7.   private static final String DEFAULT_FILE = "data.ser";
8.
9.   public static void main(String [] args){
10.    String fileName;
11.    if (args.length == 1){
12.      fileName = args[0];
13.    }
14.    else{
15.      fileName = DEFAULT_FILE;
16.    }
17.    serialize(fileName);
18.  }
19.
20.  public static void serialize(String fileName){
21.    try{
22.      serializeToFile(createData(), fileName);
23.    }
24.    catch (IOException exc){
25.      exc.printStackTrace();
26.    }
27.  }
28.
29.  private static Serializable createData(){
30.    Contact contact1 = new ContactImpl("Dennis", "Moore", "Managing Director", "Highway Man, LTD");
31.    Contact contact2 = new ContactImpl("Joseph", "Mongolfier", "High Flyer", "Lighter than Air Productions");
32.    Contact contact3 = new ContactImpl("Erik", "Njoll", "Nomad without Portfolio", "Nordic Trek, Inc.");
33.    Contact contact4 = new ContactImpl("Lemming", "", "Principal Investigator", "BDA");
34.
35.    Project project = new Project ("IslandParadise", "Acquire a personal island paradise", contact2);
36.
37.    Task task1 = new Task(project, "Fortune", "Acquire a small fortune", contact4, true);
38.    Task task2 = new Task(project, "Isle", "Locate an island for sale", null, true);
39.    Task task3 = new Task(project, "Name", "Decide on a name for the island", contact3, false);
40.    project.addProjectItem(task1);
41.    project.addProjectItem(task2);
42.    project.addProjectItem(task3);
43.
44.    Task task4 = new Task(task1, "Fortune1", "Use psychic hotline to predict winning lottery numbers", null, false);
45.    Task task5 = new Task(task1, "Fortune2", "Invest winnings to ensure 50% annual interest", contact1, true);
46.    Task task6 = new Task(task2, "Isle1", "Research whether climate is better in the Atlantic or Pacific", contact1, true);
47.    Task task7 = new Task(task2, "Isle2", "Locate an island for auction on EBay", null, false);
```

```
48.    Task task8 = new Task(task2, "Isle2a", "Negotiate for sale of the
   island", null, false);
49.    Task task9 = new Task(task3, "Name1", "Research every possible name in
   the world", null, true);
50.    Task task10 = new Task(task3, "Name2", "Eliminate any choices that are
   not coffee-related", contact4, false);
51.    task1.addProjectItem(task4);
52.    task1.addProjectItem(task5);
53.    task2.addProjectItem(task6);
54.    task2.addProjectItem(task7);
55.    task2.addProjectItem(task8);
56.    task3.addProjectItem(task9);
57.    task3.addProjectItem(task10);
58.    return project;
59. }
60.
61. private static void serializeToFile(Serializable content, String fileName)
   throws IOException{
62.    ObjectOutputStream serOut = new ObjectOutputStream(new
   FileOutputStream(fileName));
63.    serOut.writeObject(content);
64.    serOut.close ();
65. }
66.}
```

Листинг А.38. DataRetriever.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5.
6. public class DataRetriever{
7.   public static Object deserializeData(String fileName){
8.     Object returnValue = null;
9.       try{
10.          File inputFile = new File(fileName);
11.         if (inputFile.exists() && inputFile.isFile()){
12.              ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(fileName));
13.              returnValue = readIn.readObject();
14.              readIn.close();
15.          }
16.         else {
17.             System.err.println("Unable to locate the file " + fileName);
18.          }
19.      }
20.      catch (ClassNotFoundException exc){
21.          exc.printStackTrace ();
22.
23.      }
24.      catch (IOException exc){
25.          exc.printStackTrace();
26.
27.      }
28.      return returnValue;
29. }
30.}
```

Листинг А.39. RunPattern.java

```
1. import java.io.File;
2. import java.util.ArrayList;
3. import java.util.Iterator;
4. public class RunPattern{
5.   public static void main(String [] arguments){
6.     System.out.println("Example for the Chain of Responsibility pattern");
7.     System.out.println ();
8.     System.out.println("This code uses chain of responsibility to obtain");
9.     System.out.println(" the owner for a particular ProjectItem, and to");
10.    System.out.println(" build up a list of project details. In each case,");
11.    System.out.println(" a call to the appropriate getter method, getOwner");
12.    System.out.println(" or getDetails, will pass the method call up the");
13.    System.out.println(" project tree.");
14.    System.out.println("For getOwner, the call will return the first nonnull");
15.    System.out.println(" owner field encountered. For getDetails, the method");
16.    System.out.println(" will build a series of details, stopping when it");
17.    System.out.println (" reaches a ProjectItem that is designated as a");
18.    System.out.println(" primary task.");
19.     System.out.println();
20.
21.    System.out.println("Deserializing a test Project for Visitor pattern");
22.     System.out.println();
23.    if (!(new File("data.ser").exists ())){
24.      DataCreator.serialize("data.ser");
25.    }
26.    Project project = (Project)(DataRetriever.deserializeData("data.ser"));
27.
28.    System.out.println("Retrieving Owner and details for each item in the
   Project");
29.    System.out.println ();
30.    getItemInfo (project);
31. }
32.
33.  private static void getItemInfo(ProjectItem item){
34.    System.out.println("ProjectItem: " + item);
35.   System.out.println(" Owner: " + item.getOwner() ) ;
36.    System.out.println(" Details: " + item.getDetails());
37.     System.out.println();
38.    if (item.getProjectItems() != null)(
39.      Iterator subElements = item.getProjectItems().iterator();
40.      while (subElements.hasNext()){
41.           getItemInfo((ProjectItem)subElements.next());
42.      }
43.    }
44. }
45.}
```

Command

Пользователям PIM-приложения может понадобиться обновлять или модифицировать хранящуюся информацию. В данном примере показано, как с помощью шаблона Command можно обеспечить выполнение функций обновления и отмены.

В данном случае поведение некоторой команды моделируется двумя интерфейсами. Базовая операция, выполняемая командой, определяется методом `execute` интерфейса Command (листинг А.40). Интерфейс `UndoableCommand` (листинг А.41) расширяет интерфейс Command методами отмены и повтора операций.

Листинг A.40. Command.java

```
1. public interface Command{
2.   public void execute ();
3. }
```

Листинг A.41. UndoableCommand.java

```
1. public interface UndoableCommand extends Command{
2.   public void undo();
3.   public void redo();
4. }
```

Примером ситуации, в которой пользователю PIM-приложения может понадобиться команда отмены, является назначение места проведения запланированного события. Каждое событие (класс Appointment, представленный в листинге A.42) содержит, помимо описания, списка приглашенных лиц и информации о времени начала и окончания, сведения о месте проведения.

Листинг A.42. Appointment.java

```
1. import java.util.Date;
2. public class Appointment{
3.   private String reason;
4.   private Contact [] contacts;
5.   private Location location;
6.   private Date startDate;
7.   private Date endDate;
8.
9.   public Appointment(String reason, Contact!] contacts, Location location,
   Date startDate, Date endDate){
10.     this.reason = reason;
11.     this.contacts = contacts;
12.     this.location = =location;
13.     this.startDate = startDate;
14.     this.endDate = endDate;
15. }
16.
17.  public String getReason(){ return reason; }
18.  public Contact[] getContacts (){ return contacts; }
19.  public Location getLocation (){ return location; }
20.  public Date getStartDate(){ return startDate; )
21.  public Date getEndDate(){ return endDate; )
22.
23.  public void setLocation(Location location){ this.location = location; }
24.
25.  public String toString(){
26.     return "Appointment:" + "\n  Reason:  " + reason +
27.     "\n     Location:  " + location + "\n  Start;  " +
28.     startDate + "\n   End:  " + "\n";
29. }
30.}
```

Класс ChangeLocationCommand (листинг А.43) реализует интерфейс UndoableCommand и снабжает приложение логикой, необходимой для изменения места проведения запланированного события.

Листинг А.43. ChangeLocationCommand.java

```
1. public class ChangeLocationCommand implements UndoableCommand{
2.   private Appointment appointment;
3.   private Location oldLocation;
4.   private Location newLocation;
5.   private LocationEditor editor;
6.
7.   public Appointment getAppointment(){ return appointment; }
8.
9.   public void setAppointment(Appointment appointment){ this.appointment =
     appointment; }
10.  public void setLocationEditor(LocationEditor locationEditor){ editor =
     locationEditor; }
11.
12.  public void execute(){
13.    oldLocation = appointment.getLocation();
14.     newLocation = editor.getNewLocation();
15.    appointment.setLocation(newLocation) ;
16.  }
17.  public void undo(){
18.    appointment.setLocation(oldLocation) ;
19.  }
20.  public void redo(){
21.     appointment.setLocation(newLocation);
22.  }
23.}
```

Класс ChangeLocationCommand обеспечивает пользователя возможностью изменить место проведения события. Это достигается путем использования приложением метода execute класса. С помощью метода undo класс обеспечивает сохранение предыдущего значения места проведения события и позволяет пользователю восстанавливать это значение, вызвав метод undo. Наконец, класс имеет метод redo, который дает пользователю возможность повторить назначение нового места проведения, если он того пожелает.

К вспомогательным классам данного примера относится класс CommandGui (листинг А.44), который используется для создания графического пользовательского интерфейса, обеспечивающего редактирование данных о месте проведения запланированного мероприятия.

Листинг А.44. CommandGui.java

```
1. import java.awt.Container;
2. import java.awt.event.ActionListener;
3. import java.awt.event.WindowAdapter;
4. import java.awt.event.ActionEvent;
5. import java.awt.event.WindowEvent;
6. import javax.swing.BoxLayout;
7. import javax.swing.JButton;
8. import javax.swing.JComponent;
9. import javax.swing.JFrame;
```

```
10.import javax.swing.JLabel;
11.import javax.swing.JPanel;
12.import javax.swing.JTextArea;
13.import javax.swing.JTextField;
14.public class CommandGui implements ActionListener, LocationEditor{
15. private JFrame mainFrame;
16. private JTextArea display;
17. private JTextField updatedLocation;
18. private JButton update, undo, redo, exit;
19. private JPanel controlPanel, displayPanel, editorPanel;
20. private UndoableCommand command;
21. private Appointment appointment;
22.
23. public CommandGui(UndoableCommand newCommand){
24.   command = newCommand;
25. }
26.
27. public void setAppointment(Appointment newAppointment){
28.   appointment = newAppointment;
29. }
30.
31. public void createGui(){
32.   mainFrame = new JFrame("Command Pattern Example");
33.   Container content = mainFrame.getContentPane();
34.   content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
35.
36.   editorPanel = new JPanel ();
37.   editorPanel.add(new JLabel("Location") );
38.   updatedLocation = new JTextField(20);
39.   editorPanel.add(updatedLocation) ;
40.   content.add(editorPanel) ;
41.
42.   displayPanel = new JPanel ();
43.   display = new JTextArea(10, 40);
44.   display.setEditable(false);
45.   displayPanel.add(display);
46.   content.add(displayPanel);
47.
48.   controlPanel = new JPanel();
49.   update = new JButton("Update Location");
50.   undo = new JButton("Undo Location");
51.   redo = new JButton("Redo Location");
52.   exit = new JButton("Exit");
53.   controlPanel.add(update);
54.   controlPanel.add(undo) ;
55.   controlPanel.add(redo) ;
56.   controlPanel.add(exit) ;
57.   content.add(controlPanel) ;
58.
59.   update.addActionListener(this);
60.   undo.addActionListener(this);
61.   redo.addActionListener(this);
62.   exit.addActionListener(this);
63.
64.   refreshDisplay () ;
65.   mainFrame.addWindowListener(new WindowCloseManager());
66.   mainFrame.pack() ;
67.   mainFrame.setVisible(true);
68. }
69.
70. public void actionPerformed(ActionEvent evt){
71.   Object originator = evt.getSource();
72.   if (originator == update){
```

```
73.       executeCommand();
74.     }
75.     if (originator == undo){
76.       undoCommand();
77.     }
78.     if (originator == redo){
79.       redoCommand();
80.     }
81.     else if (originator == exit){
82.       exitApplication();
83.     }
84.  )
85.
86.  private class WindowCloseManager extends WindowAdapter{
87.     public void windowClosing(WindowEvent evt){
88.       exitApplication();
89.     }
90.  }
91.
92.  public Location getNewLocation(){
93.     return new LocationImpl(updatedLocation.getText());
94.  }
95.
96.  private void executeCommand (){
97.     command.execute();
98.     refreshDisplay();
99.  }
100.
101. private void undoCommand(){
102.    command.undo();
103.    refreshDisplay ();
104. }
105.
106. private void redoCommand(){
107.    command.redo();
108.    refreshDisplay();
109. }
110.
111. private void refreshDisplay(){
112.   display.setText (appointment.toString()) ;
113. }
114.
115. private void exitApplication(){
116.    System.exit(0);
117. }
118.}
```

Нетрудно заметить, что класс CommandGui реализует интерфейс LocationEditor, представленный в листинге А.45. Этот интерфейс определяет метод getNewLocation, который обеспечивает классу ChangeLocationCommand возможность получения значения нового места проведения запланированного события из графического пользовательского интерфейса.

Листинг А.45, LocationEditor.java

```
1. public interface LocationEditor{
2.    public Location getNewLocation();
3. }
```

Интерфейсы Location (листинг А.48) и Contact (листинг А.46), а также реализующие их классы LocationImpl (листинг А.49) и ContactImpl (листинг А.47) играют вспомогательную роль, представляя информацию, необходимую классу Appointment.

Листинг А.46. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitleO;
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А.47. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.   public static final String EOL_STRING =
7.      System.getProperty("line.separator");
8.
9.   public ContactImpl(){ }
10.  public ContactImpl(String newFirstName, String newLastName,
11.     String newTitle, String newOrganization){
12.       firstName = newFirstName;
13.       lastName = newLastName;
14.       title = newTitle;
15.       organization = newOrganization;
16.  }
17.
18.  public String getFirstName(){ return firstName; }
19.  public String getLastName(){ return lastName; }
20.  public String getTitle(){ return title; }
21.  public String getOrganization (){ return organization; }
22.
23.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
24.  public void setLastName(String newLastName){ lastName = newLastName; }
25.  public void setTitle(String newTitle) { title = newTitle; }
26.  public void setOrganization(String newOrganization){ organization =
     newOrganization; }
27.
28.  public String toString(){
29.     return firstName + " " + lastName;
30.  }
31.}
```

Листинг А.48. Location.java

```
1. import java.io.Serializable;
2. public interface Location extends Serializable{
3.   public String getLocation();
4.   public void setLocation(String newLocation);
5. }
```

Листинг А.49. LocationImpl.java

```
1. public class LocationImpl implements Location{
2.   private String location;
3.
4.   public LocationImpl(){ }
5.   public LocationImpl(String newLocation){
6.      location = newLocation;
7.   }
8.
9.   public String getLocation(){ return location; }
10.
11.  public void setLocation(String newLocation){ location = newLocation; }
12.
13.  public String toString(){ return location; }
14.}
```

Представленный в листинге А.50 класс RunPattern загружает данные для рассматриваемого примера и создает экземпляр класса CommandGui. Полученный пользовательский интерфейс позволяет вносить изменения в место проведения запланированного события, а также обновлять эту информацию, отменять и повторять выполненные операции.

Листинг А.50. RunPattern.java

```
1. import java.util.Calendar;
2. import java.util.Date;
3.
4. public class RunPattern{
5.   private static Calendar dateCreator = Calendar.getInstance();
6.
7.   public static void main(String [] arguments){
8.      System.out.println("Example for the Command pattern");
9.      System.out.println();
10.     System.out.println("This sample will use a command class called");
11.     System.out.println(" ChangeLocationCommand to update the location");
12.     System.out.println(" of an Appointment object.");
13.     System.out.println("The ChangeLocationCommand has the additional");
14.     System.out.println(" ability to undo and redo commands, so it can");
15.     System.out.println(" set the location back to its original value,");
16.     System.out.println(" if desired.");
17.     System.out.println();
18.
19.     System.out.println("Creating an Appointment for use in the demo");
20.    Contact [] people = { new ContactImpl() , new ContactImpl() };
21.     Appointment appointment = new Appointment("Java Twister Semi-Finals",
22.      people, new LocationImpl(""), createDate(2001, 10, 31, 14, 30),
```

```
23.    createDate(2001, 10, 31, 14, 31));
24.
25.    System.out.println("Creating the ChangeLocationCommand");
26.    ChangeLocationCommand cmd = new ChangeLocationCommand();
27.  . cmd.setAppointment(appointment)  ;
28.
29.    System.out.println("Creating the GUI");
30.    CommandGui application = new CommandGui(cmd);
31.    application.setAppointment(appointment)  ;
32.  .  cmd.setLocationEditor(application);
33.    application.createGui();
34.
35.  }
36. public static Date createDate(int year, int month, int day, int hour, int
    minute){
37.   dateCreator.set(year, month, day, hour, minute);
38.     return dateCreator.getTime();
39.  }
40. }
```

Interpreter

Основа основ шаблона Interpreter— это иерархия классов Expression. Именно она определяет грамматику, которая используется для создания и разбора выражений. Интерфейс Expression — это не только база для всех выражений. Он также определяет метод `interpret`, который и выполняет анализ выражения.

В табл. А. 1 перечислены интерфейсы, обеспечивающие построение иерархии классов Expression, и соответствующая им информация.

Таблица А.1. Назначение реализаций интерфейса Expression

Интерфейс	*Назначение*	*Листинг*
Expression	Общий интерфейс для всех выражений	А.51
ConstantExpression	Представляет константу	А.52
`VariableExpression`	Представляет переменную, полученную путем вызова метода некоторого класса	А.53
`CompoundExpression`	Пара выражений сравнения, при анализе которых получается логический результат	А.54
`AndExpression`	Логическое "И" для двух выражений	А.55
OrExpression	Логическое "ИЛИ" для двух выражений	А.56
`ComparisonExpression`	Пара выражений, при анализе которых получается логический результат	А.57
EqualsExpression	Выполняет метод **equals,** сравнивающий результаты двух выражений	А.58
`ContainsExpression`	Проверяет, содержит ли первое выражение, представленное в виде строки, вторую строку	А.59

Листинг А.51. Expression.java

```
1. public interface Expression)
2.  void interpret(Context c);
3. }
```

Листинг А.52. ConstantExpression.java

```
1. import java.lang.reflect.Method;
2. import java.lang.reflect.InvocationTargetException;
3. public class ConstantExpression implements Expression}
4.  private Object value;
5.
6.  public ConstantExpression(Object newValue){
7.    value = newValue;
8.  }
9.
10.  public void interpret(Context c){
11.    c.addVariable(this, value);
12. }
13.}
```

Листинг А.53. VariableExpression.java

```
1. import java.lang.reflect.Method;
2. import java.lang.reflect.InvocationTargetException;
3. public class VariableExpression implements Expression{
4.  private Object lookup;
5.  private String methodName;
6.
7.  public VariableExpression(Object newLookup, String newMethodName){
8.    lookup = newLookup;
9.    methodName = newMethodName;
10. }
11.
12.  public void interpret(Context c){
13.   try{
14.     Object source = c.get(lookup);
15.     if (source != null){
16.         Method method = source.getClass().getMethod(methodName, null);
17.         Object result = method.invoke(source, null);
18.          c.addVariable(this, result);
19.     }
20.   }
21.   catch (NoSuchMethotException exc){ }
22.   catch (IllegalAccessException exc){ }
23.   catch (InvocationTargetException exc){ }
24. }
25.}
```

Листинг A.54. CompoundExpression.java

```
1. public abstract class CompoundExpression implements Expression{
2.   protected ComparisonExpression expressionA;
3.   protected ComparisonExpression expressionB;
4.
5.   public CompoundExpression(ComparisonExpression expressionA,
   ComparisonExpression  expressionB){
6.      this.expressionA = expressionA;
7.      this.expressionB = expressionB;
8.   )
9. }
```

Листинг A.55. AndExpression.java

```
1. public class AndExpression extends CompoundExpression(
2.   public AndExpression(ComparisonExpression expressionA, ComparisonExpression
   expressionB) (
3.      super(expressionA, expressionB);
4. }
5.   public void interpret(Context c) {
6.      expressionA.interpret(c);
7.      expressionB.interpret(c);
8.      Boolean result = new
   Boolean(((Boolean)c.get(expressionA)).booleanValue() &&
    ((Boolean)c.get(expressionB)).booleanValue());
9.      c.addVariable(this, result);
10. }
11.}
```

Листинг A.56. OrExpression.java

```
1. public class OrExpression extends CompoundExpression{
2.   public OrExpression(ComparisonExpression expressionA, ComparisonExpression
   expressionB){
3.      super(expressionA, expressionB);
4.   }
5.   public void interpret (Context c){
6.      expressionA.interpret(c);
7.      expressionB.interpret(c) ;
8.      Boolean result = new
   Boolean(((Boolean)c.get(expressionA)).booleanValue() ||
    ((Boolean)c.get(expressionB)).booleanValue());
9.      c.addVariable(this, result);
10. }
11.}
```

Листинг A.57. ComparisonExpression.java

```
1. public abstract class ComparisonExpression implements Expression)
2.   protected Expression expressionA;
3.   protected Expression expressionB;
4.
5.   public ComparisonExpression(Expression expressionA,  Expression expressionB) {
```

```
6.    this.expressionA = expressionA;
7.    this.expressionB = expressionB;
8.  }
9. }
```

Листинг A.58. EqualsExpression.java

```
1. public class EqualsExpression extends ComparisonExpression{
2.  public EqualsExpression(Expression expressionA, Expression expressionB{
3.    super expressionA, expressionB) ;
4.  >
5.
6.  public void interpret(Context c){
7.    expressionA.interpret(c) ;
8.    expressionB.interpret(c) ;
9.    Boolean result = new Boolean
   (c.get(expressionA).equals(c.get(expressionB))) ;
10.   c.addVariable (this, result) ;
11. }
12.}
```

Листинг A.59. ContainsExpression.java

```
1. public class ContainsExpression extends ComparisonExpression{
2.  public ContainsExpression(Expression expressionA, Expression expressionB){
3.    super (expressionA, expressionB) ;
4. }
5.
6.  public void interpret (Context c){
7.    expressionA.interpret(c) ;
8.    expressionB.interpret(c) ;
9.    Object exprAResult = c.get(expressionA) ;
10.   Object exprBResult = c.get expressionB) ;
11.   if ((exprAResult instanceof String) && (exprBResult instanceof String)){
12.     if (((String)exprAResult) .indexOf((String)exprBResult) != -1){
13.         c.addVariable(this, Boolean.TRUE) ;
14.         return;
15.      }
16.    }
17.   c.addVariable(this, Boolean.FALSE) ;
18.   return;
19. }
20.)
```

Класс Context (листинг А.60) предоставляет память, необходимую для анализа выражений. Этот класс, по сути, является оболочкой (wrapper) класса `HashMap`. В данном примере объекты класса Expression предоставляют классу HashMap ключи, по которым в хеш-памяти сохраняются результаты вызова метода `interpret`.

Листинг A.60. Context.java

```
1. import java.util.HashMap;
2. public class Context{
3.   private HashMap map = new HashMap();
4.
5.   public Object get (Object name){
6.     return map.get(name) ;
7.   }
8.
9.   public void addVariable(Object name, Object value){
10.    map.put(name, value);
11.  }
12.}
```

Имея такой набор базовых выражений, можно выполнять довольно сложные операции. Рассмотрим объект класса ContactList (листинг A.61), в котором содержатся сведения о контактных лицах. У класса имеется метод getContactsMatching-Expression, который для каждого объекта Contact вычисляет результат, представленный объектом класса Expression, и возвращает объект класса ArrayList.

Листинг A.61. ContactList.java

```
1. import.java.io.Serializable;
2. import.java.util.ArrayList;
3. import.java.util.Iterator;
4. public class ContactList implements Serializable{
5.   private ArrayList contacts = new ArrayList();
6.
7.   public ArrayList getContacts(){ return contacts; }
8.   public Contact [] get ContactsAsArray(){ return (Contact []
   (contacts.toArray(newContact [1])); }
9.
10.  public ArrayList getContactsMatchingExpression(Expression expr, Context
   ctx, Object key){
11.    ArrayList results = new ArrayList();
12.    Iterator elements = contacts.iterator();
13.    while (elements.hasNext(){
14.      Object currentElement = elements.next();
15.      ctx.addVariable(key, currentElement);
16.      expr.interpret(ctx);
17.      Object interpretResult = ctx.get(expr);
18.      if ((interpretResult != null)
   &&(interpretResult.equals(Boolean.TRUE))){
19.           results.add(currentElement);
20.      }
21.    }
22.    return results;
23.  }
24.
25.  public void setContacts(ArrayList newContacts)( contacts = newContacts; }
26.
27.  public void addContact(Contact element){
28.    if (!contacts.contains(element)){
29.      contacts.add(element);
30.    }
31.  }
32.  public void removeContact(Contact element){
```

```
33.     contacts.remove(element);
34. }
35.
36. public String toString(){
37.     return contacts.toString();
38. }
39.}
```

Используя иерархию объектов класса Expression и объект класса ContactList, можно выполнять запросы на поиск объектов класса Contact к объекту класса ContactList таким образом, словно последний является базой данных. Например, можно выполнить поиск всех контактных лиц, в названии должности которых используется слово "Java". Для этого нужно проделать следующие операции.

1. Создать объект класса ConstantExpression и инициализировать его строкой "Java".
2. Создать объект класса VariableExpression и инициализировать его ссылкой на объект класса Contact и строкой "getTitle".
3. Создать объект класса ContainsExpression, передав ему в качестве первого параметра ссылку на объект класса VariableExpression, а в качестве второго — ссылку на объект класса ConstantExpression.
4. Передать ссылку на объект класса ContainsExpression методу getContactsMatchingExpression объекта класса ContactList.

Интерфейс Contact (листинг А.62) и реализующий его класс ContactImpl (листинг А.63) представляют объекты бизнес-модели, над которыми выполняются операции данного примера.

Листинг А.62. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName () ;
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization () ;
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName (String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А.63. ContactImpl.java

```
1. public class ContactImpl implements Contact)
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
```

```
6.
7.  public ContactImpl(){}
8.  public ContactImpl(String newFirstName, String newLastName,
9.    String newTitle, String newOrganization){
10.     firstName = newFirstName;
11.     lastName = newLastName;
12.     title = newTitle;
13.     organization = newOrganization;
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization; }
25.
26. public String toString(){
27.   return firstName + SPACE + lastName;
28. }
29.}
```

В данном примере показано, как на основе шаблона Interpreter можно создать механизм, обеспечивающий поиск объектов класса Contact в некой структуре, соответствующей структуре адресной книги. Нужно заметить, что классы выражений можно использовать с любыми другими классами, обеспечивая тем самым поиск любых объектов, применяющихся в PIM-приложении.

Класс RunPattern (листинг А.64) демонстрирует функциональность шаблона Interpreter, создавая экземпляр класса ContactList и применяя группу выражений сравнения к элементам соответствующего списка.

Листинг А.64. RunPattern.java

```
1. public class RunPattern{
2.  public static void main(String [] arguments){
3.    System.out.println("Example for the Interpreter pattern");
4.    System.out.println("In this demonstration, the syntax defined");
5.    System.out.println(" by the Interpreter can be used to search");
6.    System.out.println(" among a collection of Contacts, returning");
7.    System.out.println(" the subset that match the given search criteria.");
8.
9.    ContactList candidates = makeContactList ();
10.   Context ctx = new Context ();
11.
12.   System.out.println("Contents of the ContactList:");
13  .  System.out.println(candidates);
14.   System.out.println();
15.
16.   Contact testContact = new ContactImpl();
17.   VariableExpression varLName = new VariableExpression(testContact,
    "getLastName");
18.   ConstantExpression constLName = new ConstantExpression("u");
19.   ContainsExpression eqLName = new ContainsExpression(varLName,
    constLName);
```

```
20.
21.    System.out.println("Contents of the search on ContactList:");
22.    System.out.println("(search was contains 'u' in Last Name)");
23.    Object result = candidates.getContactsMatchingExpression(eqLName, ctx,
   testContact);
24.    System.out.println(result) ;
25.
26.    VariableExpression varTitle = new VariableExpression(testContact,
   "getTitle");
27.    ConstantExpression constTitle = new ConstantExpression("LT");
28.    EqualsExpression eqTitle = new EqualsExpression(varTitle, constTitle);
29.
30.    System.out.println("Contents of the search on ContactList:");
31.    System.out.println("(search was all LT personnel)");
32.    result = candidates.getContactsMatchingExpression(eqTitle, ctx,
   testContact);
33.     System.out.println(result);
34.     System.out.println();
35.
36.    VariableExpression varLastName - new VariableExpression(testContact,
   "getLastName") ;
37.    ConstantExpression constLastName = new ConstantExpression("S");
38.    ContainsExpression cLName = new ContainsExpression (varLastName,
   constLastName) ;
39.
40.    AndExpression andExpr = new AndExpression(eqTitle, cLName);
41.
42.    System.out.println("Contents of the search on ContactList:");
43.    System.out.println("(search was all LT personnel with 'S' in Last Name)");
44.    result = candidates.getContactsMatchingExpression(andExpr, ctx,
   testContact);
45.    System.out.println(result) ;
46. }
47.
48. private static ContactList makeContactList(){
49.    ContactList returnList = new ContactList();
50.    returnList.addContact(new ContactImpl("James", "Kirk", "Captain", "USS
   Enterprise"));
51.    returnList.addContact(new ContactImpl("Mr.", "Spock", "Science Officer",
   "USS Enterprise"));
52.    returnList.addContact(new ContactImpl("LT", "Uhura", "LT", "USS
   Enterprise"));
53.    returnList.addContact(new ContactImpl("LT", "Sulu", "LT", "USS
   Enterprise"));
54.    returnList.addContact(new ContactImpl("Ensign", "Checkov", "Ensign",
   "USS Enterprise"));
55.    returnList.addContact(new ContactImpl("Dr.", "McCoy", "Ship's Doctor",
   "USS Enterprise"));
56.    returnList.addContact(new ContactImpl("Montgomery", "Scott", "LT", "USS
   Enterprise"));
57.    return returnList;
58. }
59.}
```

Iterator

Данный пример построен на использовании коллекций из стандартной библиотеки языка Java. Для работы с коллекциями интерфейс java.util.Iterator определяет основные методы, предназначенные для обеспечения навигации— hasNext и next. Необходимо отметить, что интерфейс Iterator обеспечивает возможность выполнения только одного прохода по коллекции. Это означает, что если приложению понадобиться вернуться к одному из первых элементов коллекции, приложению придется создать новый экземпляр класса Iterator.

Интерфейс Iterating (листинг A.65) содержит определение лишь одного метода, названного getIterator. Этот интерфейс используется для идентификации любого класса, входящего в состав PIM-приложения, как такого, который может генерировать итератор коллекции.

Листинг A.65. Iterating.java

```
1. import java.util.Iterator;
2. import java.io.Serializable;
3. public interface Iterating extends Serializable{
4.  public Iterator getIterator ();
5. }
```

Интерфейсы ToDoList (листинг A.66) и ToDoListCollection (листинг A.67), расширяющие интерфейс Iterating, используются в данном примере для определения двух коллекций. Интерфейс ToDoList предназначен для представления классов, обеспечивающих хранение списка неотложных дел, а интерфейс ToDoListCollection — для представления коллекций классов ToDoList, хранимых PIM-приложением.

Листинг A.66. ToDoList.java

```
1. public interface ToDoList extends Iterating(
2.  public void add(String item);
3.  public void add(String item, int position);
4.  public void remove(String item);
5.  public int getNumberOfItems();
6.  public String getListName();
1.  public void setListName(String newListName);
8. )
```

Листинг A.67. ToDoListCollection.java

```
1. public interface ToDoListCollection extends Iterating{
2.  public void add(ToDoList list);
3.  public void remove(ToDoList list);
4.  public int getNumberOf Items ();
5. }
```

Приведенные выше интерфейсы реализуются, соответственно, классами ToDoListImpl и ToDoListCollectionImpl. Класс ToDoListImpl (листинг А.68) обеспечивает хранение своих элементов путем использования класса ArrayList, который позволяет упорядочивать элементы по абсолютным значениям, а также допускает дублирование элементов. Класс ToDoListCollectionImpl (листинг А.69) использует класс HashTable, который не поддерживает упорядочения и сохраняет все элементы в виде пар "ключ — значение". Хотя, как видно, поведение коллекций весьма различно, у обоих имеются итераторы, обеспечивающие навигацию по коллекциям.

Листинг А.68. ToDoListImpl.java

```
1. import java.util.Iterator;
2. import java.util.ArrayList;
3. public class ToDoListImpl implements ToDoList{
4.   private String listName;
5.   private ArrayList items = new ArrayList ();
6.
7.   public void add(String item){
8.     if (!items.contains(item)){
9.       items.add(item);
10.    }
11. }
12. public void add(String item, int position){
13.    if (!items.contains(item)){
14.      items.add(position, item);
15.    }
16. }
17. public void remove(String item){
18.    if (items.contains(item)){
19.      items.remove(items.indexOf(item));
20.    }
21. }
22..
23.  public int getNumberOfItems() { return items.size();}
24.  public Iterator getIterator(){ return items.iterator(); }
25.  public String getListName(){ return listName;}
26.  public void setListName(String newListName){ listName = newListName; }
27.
28.  public String toString(){ return listName;}
29.}
```

Листинг А.69. ToDoListCollectionImpl.java

```
1. import java.util.Iterator;
2. import java.util.HashMap;
3. public class ToDoListCollectionImpl implements ToDoListCollection{
4.   private HashMap lists = new HashMap ();
5.
6.   public void add(ToDoList list){
7.     if (!lists.containsKey(list.getListName())){
8.       lists.put(list.getListName(), list);
9.     }
10. }
11. public void remove(ToDoList list){
12.    if (lists.containsKey(list.getListName())){
13.      lists.remove(list.getListName() ) ;
```

```
14.   }
15.  }
16.  public int getNumberOfItems(){ return lists.size(); )
17.  public Iterator getIterator(){ return lists.values().iterator(); }
18.  public String toString(){ return getClass().toString(); )
19.}
```

Так как оба класса предоставляют доступ к итератору, достаточно просто написать код, который обеспечивает перемещение по элементам соответствующих коллекций. Так, на примере класса `ListPrinter` (листинг А.70) показано, как использовать итератор для вывода на печать содержимого коллекций, представленного в строковом виде. Класс имеет три метода: `printToDoList`, `printToDoListCollection` и `printIteratingElement`. В каждом из методов последовательный перебор всех элементов коллекций организован с использованием очень простого цикла `while`.

Листинг А.70. ListPrinter.java

```
1. import java.util.Iterator;
2. import java.io.PrintStream;
3. public class ListPrinter{
4.   public static void printToDoList(ToDoList list, PrintStream output){
5.     Iterator elements = list.getIterator();
6.    output.println(" List - "+ list + " : " ) ;
7.     while (elements.hasNext()){
8.       output.println("\t" +elements.next());
9.     }
10.  }
11.
12.  public static void printToDoListCollection(ToDoListCollection iotsOfLists,
   PrintStream output){
13.     Iterator elements = lotsOfLists.getIterator();
14.    output.println("\"To Do\" ListCollection:");
15.     while (elements.hasNext()){
16.      printToDoList((ToDoList)elements.next(), output);
17.    }
18.  }
19.
20.  public static void printIteratingElement(Iterating element, PrintStream
   output){
21.     output.println("Printing the element " + element);
22.     Iterator elements = element.getIterator();
23.     while (elements.hasNext()){
24.      Object currentElement = elements.next();
25.       if (currentElement instanceof Iterating){
26.           printIteratingElement((Iterating)currentElement, output);
27.            output.println();
28.      }
29.      else{
30.           output.println(currentElement);
31.      }
32.    }
33.  }
34.)
```

Мощь комбинации шаблона Iterator и полиморфизма лучше всего видна на примере метода printIteratingElement. В нем, в частности, показано, как распечатать в строковом представлении данные любого класса, реализующего интерфейс Iterating. При этом методу совершенно не нужно ничего знать о структуре нижележащей коллекции за исключением того, что она может генерировать итераторы.

В данном примере используются два вспомогательных класса DataCreator и DataRetriever, исходный код которых содержится в листингах А.71 и А.72, соответственно.

Листинг А.71. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. public class DataCreator{
6.   private static final String DEFAULT_FILE = "data.ser";
7.
8.   public static void main(String [] args){
9.     String fileName;
10.    if (args.length == 1){
11.      fileName = args[0];
12.    }else{
13.      fileName = DEFAULT_FILE;
14.    }
15.    serialize(fileName);
16. }
17.
18.  public static void serialize(String fileName){
19.    try{
20.      serializeToFile(createData(), fileName);
21.    } catch (IOException exc) {
22.      exc.printStackTrace() ;
23.    }
24. >
25.
26.  private static Serializable createData (){
27.    ToDoListCollection data = new ToDoListCollectionImpl();
28.    ToDoList listOne = new ToDoListImpl();
29.    ToDoList listTwo = new ToDoListImpl();
30.    ToDoList listThree = new ToDoListImpl();
31.    listOne.setListName("Daily Routine");
32.    listTwo.setListName("Programmer hair washing procedure");
33.    listThree.setListName("Reading List");
34.    listOne.add("Get up (harder some days than others)");
35.    listOne.add("Brew cuppa Java");
36.    listOne.add("Read JVM Times");
37.    listTwo. add("Lather") ;
38.    listTwo.add("Rinse");
39.    listTwo.add("Repeat");
40.    listTwo.add("(eventually throw a TooMuchHairConditioner exception)");
41.    listThree.add("The complete annotated aphorisms of Duke");
42.    listThree.add("How green was my Java");
43.    listThree.add("URL, sweet URL");
44.    data.add(listOne);
45.    data.add(listTwo);
46.    data.add(listThree);
47.    return data;
48. }
```

```
49.
50.  private static void serializeToFile(Serializable data, String fileName)
     throws IOException{
51.    ObjectOutputStream serOut = new ObjectOutputStream(new
     FileOutputStream(fileName));
52.    serOut.writeObject(data);
53.    serOut.close();
54.  }
55. }
```

Листинг A.72. DataRetriever.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5.
6. public class DataRetriever{
7.   public static Object deserializeData(String fileName){
8.     Object returnValue = null;
9.     try{
10.      File inputFile = new File (fileName);
11.      if (inputFile.exists() && inputFile.isFile()){
12.          ObjectInputStream readIn = new ObjectInputStream(new
     FileInputStream(fileName)) ;
13.           returnValue = readIn.readObject();
14.           readIn.close();
15.       )elsef
16.           System.err.println("Unable to locate the file " + fileName);
17.       }
18.     }catch (ClassNotFoundException exc) {
19.       exc.printStackTrace();
20.     '}catch (IOException exc){
21.       exc.printStackTrace();
22.     )
23.     return returnValue;
24.  }
25. }
```

Работу классов, используемых в данном примере, координирует класс RunPattern, представленный в листинге A.73. Этот класс загружает тестовые данные, а затем вызывает метод `printToDoListCollection` класса `ListPrinter` для вывода содержимого всех списков.

Листинг A.73. RunPattern.java

```
1. import java.io.File;
2. import java.io.IOException;
3. public class RunPattern{
4.   public static void main(String [] arguments){
5.     System.out.println("Example for the Iterator pattern");
6.     System.out.println(" This code sample demonstrates how an Iterator can
     enforce");
7.     System.out.println(" uniformity of processing for different collection
     types.");
```

```
8.    System.out.println(" In this case, there are two classes, ToDoListImpl
   and");
9.    System.out.println (" ToDoListCollectionImpl, that have different storage
   needs.") ;
10.   System.out.println(" ToDoListImpl uses an ArrayList to store its
   elements in");
11.   System.out.println(" ordered form. The ToDoListCollectionImpl uses a
   HashMap,") ;
12.   System. out .println (" since it must differentiate between ToDoListImpl
   objects by");
13.   System.out.println(" their String identifiers.");
14.   System.out.println();
15.   System.out.println("Although the two classes use different underlying
   collections,");
16.   System.out.println(" the ListPrinter class can use the Iterator produced
   by each");
17.   System.out.println (" to print out a set of list contents.");
18.   System.out.println();
19.
20.   if (!(new File("data.ser").exists())){
21.   DataCreator.serialize("data. ser") ;
22.   }
23.   ToDoListCollection lists =
   (ToDoListCollection)(DataRetriever.deserializeData("data.ser"));
24.
25.   System.out.println("Lists retrieved. Printing out contents using the
   Iterator");
26.   System.out.println();
27.   ListPrinter.printToDoListCollection(lists, System.out);
28. }
29.}
```

Mediator

В данном примере шаблон Mediator используется для управления взаимодействием панелей графического пользовательского интерфейса. В базовом варианте данного GUI имеется панель, позволяющая выбрать контактное лицо из списка, панель, обеспечивающая редактирование, и панель, отображающая текущее состояние контакта. Посредник взаимодействует с каждой панелью, вызывая методы, которые предназначены для обновления элементов GUI в соответствии со вносимыми изменениями.

Класс `MediatorGui` (листинг А.74) создает в приложении главное окно и три панели. Кроме того, он создает объект-посредник и связывает его с тремя дочерними панелями.

Листинг А.74. MediatorGui.java

```
1. import java.awt.Container;
2. import java.awt.event.WindowEvent;
3. import java.awt.event.WindowAdapter;
4. import javax.swing.BoxLayout;
5. import javax.swing.JButton;
6. import javax.swing.JFrame;
7. import javax.swing.JPanel ;
8. public class MediatorGui{
9.   private ContactMediator mediator;
10.
```

```
11. public void setContactMediator(ContactMediator newMediator){ mediator =
    newMediator; }
12.
13. public void createGui(){
14.   JFrame mainFrame = new JFrame("Mediator example");
15.   Container content = mainFrame.getContentPane();
16.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
17.    ContactSelectorPanel select = new ContactSelectorPanel(mediator);
18.    ContactDisplayPanel display = new ContactDisplayPanel(mediator);
19.    ContactEditorPanel edit = new ContactEditorPanel(mediator);
20.     content.add(select);
21.     content.add(display);
22.  .  content.add(edit);
23. . mediator.setContactSelectorPanel(select) ;
24.  .  mediator.setContactDisplayPanel(display);
25.  .  mediator.setContactEditorPanel(edit);
26.     mainFrame.addWindowListener(new WindowCloseManager());
27.     mainFrame.pack();
28.    mainFrame.setVisible(true) ;
29.  }
30. private class WindowCloseManager extends WindowAdapter{
31.    public void windowClosing(WindowEvent evt){
32.       System.exit(0);
33.     }
34.  }
35. }
```

Самой простой панелью GUI является панель, представленная объектом класса ContactDisplayPanel (листинг А.75). В этом классе имеется метод, названный contactChanged, который обновляет представление объекта на экране в соответствии с новыми значениями параметра Contact.

Листинг А.75. ContactDisplayPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JScrollPane;
4. import javax.swing.JTextArea:
5. public class ContactDisplayPanel extends JPanel{
6.    private ContactMediator mediator;
7.    private JTextArea displayRegion;
8.
9.    public ContactDisplayPanel(){
10.      createGui();
11. )
12.   public ContactDisplayPanel(ContactMediator newMediator){
13.     setContactMediator(newMediator) ;
14.     createGui () ;
15. }
16.   public void createGui(){
17.      setLayout(new BorderLayout());
18.      displayRegion = new JTextArea(10, 40);
19.     displayRegion.setEditable(false) ;
20.      add(new JScrollPane(displayRegion));
21. }
22.   public void contactChanged(Contact contact){
23.      diplayRegion.setText(
24.        "Contact\n\tName: " + contact.getFirstName() +
25.        " " + contact.getLastName() + "\n\tTitle;" +
26.        contact.getTitle() + "\n\tOrganization: " +
```

```
27.       contact.getOrganization());
28.  }
29.  public void setContactMediator(ContactMediator newMediator){
30.     mediator = newMediator;
31.  }
32.}
```

Класс ContactSelectorPanel (листинг А.76) позволяет пользователю выбирать для отображения и редактирования объект класса Contact.

Листинг А.76. ContactSelectorPanel.java

```
1. import java.awt.event.ActionEvent;
2. import java.awt.event.ActionListener;
3. import javax.swing.JComboBox;
4. import javax.swing.JPanel;
5.
6. public class ContactSelectorPanel extends JPanel implements ActionListener{
7.   private ContactMediator mediator;
8.   private JComboBox selector;
9.
10.  public ContactSelectorPanel(){
11.     createGui();
12.  }
13.  public ContactSelectorPanel(ContactMediator newMediator){
14.     setContactMediator(newMediator);
15.     createGui;
16.  }
17.
18.  public void createGui(){
19.     selector = new JComboBox (mediator.getAllContacts());
20.     selector.addActionListener(this) ;
21.     add(selector) ;
22.  }
23.
24.  public void actionPerformed(ActionEvent evt){
25.     mediator.selectContact((Contact)selector.getSelectedItem());
26.  }
27.  public void addContact(Contact contact){
28.     selector.addItem(contact);
29.     selector.setSelectedItem(contact);
30.  }
31.  public void setContactMediator(ContactMediator newMediator){
32.     mediator = newMediator;
33.  }
34.}
```

Класс ContactEditorPanel (листинг А.77) предоставляет интерфейс для редактирования полей выбранного контакта.

Листинг А.77. ContactEditorPanel.java

```
1. import java.awt.BorderLayout;
2. import java.awt.GridLayout;
3. import java.awt.event.ActionEvent;
4. import java.awt.event.ActionListener;
5. import javax.swing.JButton;
```

```
6. import javax.swing.JLabel;
7. import javax.swing.JPanel;
8. import javax.swing.JTextField;
9. public class ContactEditorPanel extends JPanel implements ActionListener{
10.  private ContactMediator mediator;
11.  private JTextField firstName, lastName, title, organization;
12.  private JButton create, update;
13.
14.  public ContactEditorPanel(){
15.     createGui()
16.  }
17.  public ContactMediator(newMediator);
18.     setContactMediator(newMediator);
19.   createGui ();
20.  }
21.  public void createGui (){
22.     setLayout(new BorderLayout());
23.
24.    JPanel editor = new JPanel();
25.    editor.setLayout(new GridLayout(4, 2));
26.    editor.add(new JLabel("First Name:"));
27.    firstName = new JTextField(20);
28.    editor.add(firstName) ;
29.    editor.add(new JLabel("Last Name:"));
30.    lastName = new JTextField(20) ;
31.    editor.add(lastName) ;
32.    editor.add(new JLabel("Title:"));
33.    title = new JTextField (20) ;
34.    editor.add(title) ;
35.    editor.add(new JLabel("Organization: "));
36.    organization = new JTextField(20) ;
37.     editor.add(organization);
38.     add(editor, BorderLayout.CENTER);
39.
40.    JPanel control = new JPanel();
41.    create = new JButton("Create Contact");
42.    update = new JButton("Update Contact");
43.     create.addActionListener(this);
44.     update.addActionListener(this);
45.     control.add(create);
46.     control.add(update);
47.    add(control, BorderLayout.SOUTH);
48.  }
49.  public void actionPerformed(ActionEvent evt){
50.    Object source = evt.getSource();
51.    if (source == create){
52.       createContact();
53.     }
54.     else if (source == update){
55.       updateContact() ;
56.     }
57.  )
58.
59.  public void createContact(){
60.     mediator.createContact(firstName.getText(), lastName.getText(),
61.       title.getText(), organization.getText());
62.  }
63.  public void updateContact(){
64.     mediator.updateContact(firstName.getText(), lastName.getText(),
65.       title.getText(), organization.getText());
66.  }
67.
68.  public void setContactFields(Contact contact){
```

```
69.    firstName.setText(contact.getFirstName());
70.    lastName.setText(contact.getLastName());
71.    title.setText(contact.getTitle());
72.    organization.setText(contact.getOrganization());
73. }
74.  public void  setContactMediator(ContactMediator newMediator){
75.     mediator = newMediator;
76. }
77.}
```

Интерфейс ContactMediator (листинг А.78) определяет набор методов для каждого компонента GUI и для методов, реализующих бизнес-правила (createContact, updateContact, select-Contact и getAllContacts).

Листинг А.78. ContactMediator.java

```
1. public interface ContactMediator{
2.   public void setContactDisplayPanel(ContactDisplayPanel displayPanel);
3.   public void setContactEditorPanel(ContactEditorPanel editorPanel);
4.   public void setContactSelectorPanel(ContactSelectorPanel selectorPanel);
5.   public void createContact(String firstName, String lastName, String title,
   String organization);
6.   public void updateContact(String firstName, String lastName, String title,
   String organization);
7.  public Contact [j getAllContacts();
8.   public void selectContact(Contact contact);
9. }
```

Реализация интерфейса ContactMediator представлена классом ContactMediator-Impl (листинг А.79). Этот класс обеспечивает выбор контакта, а также содержит методы, посредством которых панели извещаются о внесении изменений пользовательского интерфейса.

Листинг А.79. ContactMediatorImpl.java

```
1. import java.util.ArrayList;
2. public class ContactMediatorlmpl implements ContactMediator{
3.   private ContactDisplayPanel display;
4.   private ContactEditorPanel editor:
5.   private ContactSelectorPanel selector;
6.  private ArrayList contacts = new ArrayList();
7.   private int contactIndex;
8.
9.   public void setContactDisplayPanel(ContactDisplayPanel displayPanel){
1C.     display = displayPanel;
11.  }
12.   public void setContactEditorPanel(ContactEditorPanel editorPanel){
13.     editor = editorPanel;
14.  }
15.   public void setContactSelectorPanel(ContactSelectorPanel){
16.     selector = selectorPanel;
17.  }
18.
19.   public void createContact(String firstName, String lastName, String title,
   String organization)(
```

```
20.   Contact newContact = new ContactImpl(firstName, lastName, title,
   organization);
21.     addContact(newContact);
22.   . selector.addContact(newContact);
23.  . display.contactChanged(newContact) ;
24.  }
25.  public void updateContact(string firstName, String lastName, String title,
   String organization){
26.    Contact updateContact = (Contact)contacts.get(contactIndex) ;
27.    if (updateContact != null){
28.  .   updateContact.setFirstName(firstName);
29.      updateContact.setLastName(lastName);
30.  .  updateContact.setTitle(title) ;
31.     updateContact.setOrganization(organization) ;
32.  .   display.contactChanged(updateContact);
33.  .  }
34.  )
35.  public void selectContact(Contact contact){
36.    if (contacts.contains(contact)){
37.      contactIndex = contacts.indexOf(contact)
38.      display.contactChanged(contact)
39.     editor.setContactFields(contact) ;
40.   }
41.  }
42. public Contact [] getAllContacts () {
43.    return (Contact []) contacts.toArray(new Contact[1]);
44.  }
45.  public void addContact(Contact contact){
46.    if (!contacts.contains(contact)){
47.      contacts.add(contact);
48.   }
49.  }
50.}
```

Класс ContactMediatorImpl взаимодействует с каждым из классов панелей по-разному. Так, обращаясь к классу ContactDisplayPanel, посредник для выполнения операций создания, обновления и выбора вызывает метод contactChanged. Обращаясь к классу ContactSelectorPanel, посредник предоставляет ему список контактов и метод getAllContacts, получает от него извещения о выбранном контакте и помещает новый объект класса Contact на панель при ее создании. Объект класса ContactEditorPanel вызывает при обращении к посреднику методы создания и обновления, а объект класса ContactSelectorPanel извещает посредника о работе панели выбора.

Вспомогательные классы в данном примере представлены классами Contact (листинг А.80) и ContactImpl (листинг А.81).

Листинг А.80. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.  public String getTitleO;
7.  public String getOrganization() ;
8.
9.   public void setFirstName(String newFirstName);
```

```
10. public void setLastName(String newLastName);
11. public void setTitle(String newTitle);
12. public void setOrganization(String newOrganization);
13.)
```

Листинг А.81. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.      String newTitle, String newOrganization){
10.       firstName = newFirstName;
11.       lastName = newLastName;
12.       title = newTitle;
13.       organization = newOrganization;
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization (){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; )
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization; }
25.
26. public String toString(){
27.    return firstName + SPACE + lastName;
28. )
29.}
```

Класс RunPattern (листинг А.82) создает графический пользовательский интерфейс и его посредника, а затем загружает тестовые данные примера.

Листинг А.82. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.      System.out.println("Example for the Mediator pattern");
4.      System.out.println("In this demonstration, the ContactMediatorImpl class
    will");
5.      System.out.println(" coordinate updates between three controls in a GUI
    - the");
6.      System.out.println(" ContactDisplayPanel, the ContactEditorPanel, and the");
7.      System.out.println(" ContactSelectorPanel. As its name suggests, the
    Mediator");
8.      System.out.println(" mediates the activity between the elements of the GUI,");
9.      System.out.println(" translating method calls from one panel into the
    appropriate");
10.     System.out.println(" method calls on the other GUI components.");
```

```
11.
12.  Contact contact = new ContactImpl("", "", "", "");
13.  Contact contact1 = new ContactImpl("Duke", "", "Java Advocate", "The
   Patterns Guild");
14.   ContactMediatorImpl mediator = new ContactMediatorImpl();
15.   mediator.addContact(contact) ;
16 . mediator.addContact(contact1) ;
17.   MediatorGui gui = new MediatorGui();
18 . gui.setContactMediator(mediator) ;
19.    gui.createGui();
20.  }
21.}
```

Memento

Практически все подсистемы PIM-приложения сохраняют в том или ином объеме информацию о своем состоянии. Эту информацию можно сохранять с помощью шаблона Memento, как показано в данном разделе на примере адресной книги. Класс `AddressBook` (листинг А.83) представляет собой коллекцию адресов, что делает его одним из кандидатов на сохранение информации о состоянии.

Листинг А.83. AddressBook.java

```
1. import java.util.ArrayList;
2. public class AddressBook{
3.   private ArrayList contacts = new ArrayList ();
4.
5.   public Object getMemento(){
6.      return new AddressBookMemento(contacts);
7.   }
8.   public void setMemento(Object object){
9.      if (object instanceof AddressBookMemento){
10.       AddressBookMemento memento =  (AddressBookMemento)object;
11.       contacts = memento.state;
12.     )
13.  }
14.
15.  private class AddressBookMemento{
16.     private ArrayList state;
17.
18.     private AddressBookMemento(ArrayList contacts){
19.       this.state = contacts;
20.     }
21.  }
22.
23.  public AddressBook(){ }
24.  public AddressBook(ArrayList newContacts){
25.     contacts = newContacts;
26.  }
27.
28.  public void addContact(Contact contact){
29.     if (!contacts.contains(contact)){
30.       contacts.add(contact) ;
31.     }
32.  }
33.  public void removeContact(Contact contact){
34.     contacts.remove(contact);
35.  }
```

```
36. public void removeAllContacts () {
37.    contacts = new ArrayList ();
38.  }
39. public ArrayList getContacts (){
40.    return contacts;
41.  }
42. public String toString(){
43.    return contacts.toString();
44.  }
45.}
```

Для сохранения информации о состоянии класса AddressBook, представленной в виде внутреннего объекта ArrayList, который предназначен для хранения объектов Address, используется внутренний по отношению к нему класс AddressBookMemento. Доступ к объекту-контейнеру можно получить, используя методы getMemento и setMemento класса AddressBook. Необходимо отметить, что класс AddressBookMemento объявлен как закрытый внутренний класс, состоящий лишь из одного закрытого конструктора. Это делает невозможным другим объектам использовать информацию о состоянии, хранящуюся в контейнере, или модифицировать ее даже в тех случаях, когда объект AddressBookMemento сохраняется за пределами класса AddressBook. Данное свойство класса AddressBookMemento полностью соответствует цели, стоящей перед шаблоном Memento: генерация объекта, содержащего "моментальный снимок" состояния, который не может быть модифицирован ни одним посторонним объектом системы.

Используемые в данном примере вспомогательные классы представляют прикладные объекты контактов, хранящихся в адресной книге, и соответствующих им адресов. Поведение этих объектов определяется интерфейсами Address (листинг А.84) и Contact (листинг А.86), а реализация этого поведения обеспечивается классами AddressImpl (листинг А.85) и ContactImpl (листинг А.87).

Листинг А.84. Address.java

```
1. import java.io.Serializable;
2. public interface Address extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.   public static final String COMMA = ",";
6.   public String getType();
7.   public String getDescription();
8.  public String getStreet();
9.   public String getCityO;
10.  public String getStateO;
11.  public String getZipCode();
12.
13.  public void setType(String newType);
14.  public void setDescription(String newDescription);
15.  public void setStreet(String newStreet);
16.  public void setCity(String newCity) ;
17.  public void setState(String newState);
18.  public void setZipCode(String newZip);
19.}
```

Листинг A.85. AddressImpl.java

```
1. public class AddressImpl implements Address{
2.   private String type;
3.   private String description;
4.   private String street;
5.   private String city;
6.   private String state;
7.   private String zipCode;
8.
9.   public AddressImpl(){ }
10.  public AddressImpl(String newDescription, String newStreet,
11.     String newCity, String newState, String newZipCode){
12.     description = newDescription;
13.     street = newStreet;
14.     city = newCity;
15.     state = newState;
16.     zipCode = newZipCode;
17. }
18.
19.  public String getType(){ return type; }
20.  public String getDescription(){ return description; }
21.  public String getStreet(){ return street; }
22.  public String get.City(){ return city; }
23.  public String getstate(){ return state; }
24.  public String getZipCode(){ return zipCode; }
25.
26.  public void setType(String newType){ type = newType; }
27.  public void setDescription(String newDescription){ description =
   newDescription;}
28.  public void setStreet(String newStreet){ street = newStreet; }
29.  public void setCity(String newCity){ city = newCity; }
30.  public void setState(String newState){ state = newState; }
31.  public void setZipCode(String newZip){ zipCode = newZip; }
32.
33.  public String toString(){
34.     return street + EOL_STRING + city + COMMA + SPACE +
35.       state + SPACE + zipCode + EOL_STRING;
36. }
37.}
```

Листинг A.86. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName() ;
6.   public String getTitle();
7.   public String getOrganization() ;
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А.87. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.   private Address address;
7.
8.   public ContactImpl(){}
9.   public ContactImpl(String newFirstName, String newLastName,
10.     String newTitle, String newOrganization, Address newAddress){
11.       firstName = newFirstName;
12.       lastName = newLastName;
13.       title = newTitle;
14.       organization = newOrganization;
15.       address = newAddress;
16. }
17.
18. public String getFirstName(){ return firstName; }
19. public String getLastName(){ return lastName; }
20. public String getTitle(){ return title; }
21. public String getOrganization(){ return organization; }
22. public Address getAddress.() { return address; }
23.
24. public void setFirstName(String newFirstName){ firstName = newFirstName; }
25. public void setLastName(String newLastName){ lastName = newLastName; }
26. public void setTitle(String newTitle){ title = newTitle; }
27. public void setOrganization(String newOrganization)) organization =
    newOrganization;}
28. public void setAddress(Address newAddress){ address = newAddress; }
29.
30. public String toString(){
31.     return firstName + " " + lastName;
32. )
33.}
```

Демонстрация использования шаблона Memento осуществляется классом RunPattern (листинг А.88), который создает адресную книгу и заносит в нее данные нескольких человек. Затем класс RunPattern сохраняет состояние этой группы контактов в объекте класса AddressBookMemento и создает дополнительный перечень контактных лиц. Последняя операция, выполняемая классом RunPattern, состоит в восстановлении исходного состояния адресной книги и осуществляется путем вызова метода setMemento соответствующего объекта.

Листинг А.88. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.   System.out.println("Example for the Memento pattern");
4.    System.out.println();
5.   System.out.println("This example will use the AddressBook to demonstrate");
6.   System.out.println(" how a Memento can be used to save and restore state.");
7.   System.out.println("The AddressBook has an inner class,
    AddressBookMemento,") ;
8.   System.out.println(" that is used to store the AddressBook state... in this");
9.   System.out.println(" case, its internal list of contacts.");
```

```
10.  System.out.println() ;
11.  System.out.println("Creating the AddressBook");
12.  AddressBook book = new AddressBook ();
13.
14.  System.out.println("Adding Contact entries for the AddressBook");
15.  book.addContact(new ContactImpl("Peter", "Taggart", "Commander", "NSEA
16.Protector", new AddressImpl()));
     book.addContact(new ContactImpl("Tawny", "Madison", "Lieutenant", "NSEA
17.Protector", new AddressImpl()));
     book.addContact(new ContactImpl("Dr.", "Lazarus", "Dr.", "NSEA Protector",
18.new AddressImpl()));
     book.addContact(new ContactImpl("Tech Sargent", "Chen", "Tech Sargent",
19."NSEA Protector", new AddressImpl ()));
20.
21.  System.out.println("Contacts added. Current Contact list:");
22.  System.out.println(book);
23.  System.out.println();
24.
25.  System.out.println("Creating a Memento for the address book");
26.  Object memento = book.getMemento();
27.  System.out.println("Now that a Memento exists, it can be used to restore");
28.  System.out.println(" the state of this AddressBook object, or to set the");
29.  System.out.println(" state of a new AddressBook.");
30.  System.out.println();
31.
32.  System.out.println("Creating new entries for the AddressBook");
33.  book.removeAllContacts();
34.  book.addContact(new ContactImpl("Jason", "Nesmith", "", "Actor's Guild",
     new AddressImpl())) ;
35.  book.addContact(new ContactImpl("Gwen", "DeMarco", "", "Actor's Guild", new
     AddressImpl())) ;
36.  book.addContact(new ContactImpl("Alexander", "Dane", "", "Actor's Guild",
     new AddressImpl ()));
37.  book.addContact(new ContactImpl("Fred", "Kwan", "", "Actor's Guild", new
     AddressImpl())) ;
38.
39.  System.out.println("New Contacts added. Current Contact list:");
40.  System.out.println(book);
41.  System.out.println();
42.  System.out.println("Using the Memento object to restore the AddressBook");
43.  System.out.println(" to its original state.");
44.  book.setMemento(memento) ;
45.  System.out.println("AddressBook restored. Current Contact list:");
46.  System.out.println(book);
47.
48.  }
49.}
```

Observer

В данном примере показано, как наблюдаемый объект рассылает всем наблюдателям информацию об обновленном состоянии объекта `Task`.

Необходимо отметить, что в любом программном коде GUI Java для обработки сообщений обычно используется именно шаблон Observer. Разрабатывая класс, реализующий интерфейс вида `ActionListener`, вы тем самым создаете класс наблюдателя. Регистрация этого класса в компоненте с помощью метода `addActionListener` связывает наблюдателя с наблюдаемым элементом, роль которого выполняет компонент GUI Java.

В данном примере наблюдаемый элемент представлен классом Task, который модифицируется GUI. Класс TaskChangeObservable (листинг А.89) отслеживает изменения в объекте класса Task с помощью методов addTaskChangeObserver и removeTaskChangeObserver.

Листинг А.89. TaskChangeObservable.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class TaskChangeObservable{
4.   private ArrayList observers = new ArrayList ();
5.
6.   public void addTaskChangeObserver(TaskChangeObserver observer){
7.     if ( !observers.contains(observer)) (
8.       observers.add(observer) ;
9.     }
10. }
11.  public void removeTaskChangeObserver(TaskChangeObserver observer){
12.    observers.remove(observer);
13. }
14.
15.  public void selectTask(Task task){
16.    Iterator elements = observers.iterator ();
17.    while (elements.hasNext()){
18.      ((TaskChangeObserver)elements.next()) .taskSelected(task);
19.    }
20. }
21.  public void addTask(Task task){
22.    Iterator elements = observers.iterator ();
23.    while (elements.hasNext()){
24.      ((TaskChangeObserver)elements.next()) .taskAdded(task);
25.    }
26. }
27.  public void updateTask(Task task){
28.    Iterator elements = observers.iterator ();
29.    while (elements.hasNext()){
30.      ((TaskChangeObserver) elements.next ()).taskChanged(task);
31.    }
32. }
33.}
```

Класс TaskChangeObservable содержит методы selectTask, updateTask и addTask. Эти методы отвечают за отправку извещения о любых изменениях классу Task.

Каждый наблюдатель должен реализовать интерфейс TaskChangeObserver (листинг А.90), чтобы класс TaskChangeObservable мог вызвать соответствующий метод наблюдателя. Если клиенту понадобится, например, вызвать метод addTask класса TaskChangeObservable, наблюдаемый объект будет последовательно вызывать методы taskAdded всех своих наблюдателей.

Листинг А.90. TaskChangeObservable.java

```
1.  public interface TaskChangeObserver{
2.  public void taskAdded(Task task);
3.  public void taskChanged(Task task);
4.  public void taskSelected(Task task);
5.  }
```

Класс ObserverGui (листинг A.91) предназначен в рассматриваемом примере для организации графического пользовательского интерфейса. Именно он создает объект класса TaskChangeObservable. Кроме того, он создает три панели TaskEditorPanel (листинг A.92), TaskHistoryPanel (листинг A.93) и TaskSelectorPanel (листинг A.94), которые реализуют интерфейс TaskChangeObserver, и связывает с ними объект класса TaskChangeObservable. После такого связывания последний получает возможность эффективно отправлять сообщения об обновлении всем трем панелям GUI.

Листинг A.91. ObserverGui.java

```
1. import java.awt.Container;
2. import java.awt.event.WindowAdapter;
3. import java.awt.event.WindowEvent;
4. import javax.swing.BoxLayout;
5. import javax.swing.JFrame;
6. public class ObserverGui{
7.   public void createGui(){
8.     JFrame mainFrame = new JFrame("Observer Pattern Example");
9.     Container content = mainFrame.getContentPane();
10.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
11.    TaskChangeObservable observable = new TaskChangeObservable();
12.    TaskSelectorPanel select = new TaskSelectorPanel(observable);
13.    TaskHistoryPanel history = new TaskHistoryPanel();
14.    TaskEditorPanel edit = new TaskEditorPanel(observable);
15.    observable.addTaskChangeObserver(select);
16.    observable.addTaskChangeObserver(history);
17.    observable.addTaskChangeObserver(edit);
18.    observable.addTask(new TaskO);
19.    content.add(select);
20.    content.add(history);
21.    content.add(edit);
22.    mainFrame.addWindowListener(new WindowCloseManager());
23.    mainFrame.pack():
24.    mainFrame.setVisible(true);
25. }
26.
27.  private class WindowCloseManager extends WindowAdapter{
28.    public void windowClosing(WindowEvent evt){
29.      System.exit(0);
30.    }
31. }
32.}
```

Листинг A.92. TaskEditorPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JLabel;
4. import javax.swing.JTextField;
5. import javax.swing.JButton;
6. import java.awt.event.ActionEvent;
7. import java.awt.event.ActionListener;
8. import java.awt.GridLayout;
9. public class TaskEditorPanel extends JPanel implements ActionListener,
   TaskChangeObserver{
10.  private JPanel controlPanel, editPanel;
```

```
11. private JButton add, update, exit;
12. private JTextField taskName, taskNotes, taskTime;
13. private TaskChangeObservable notifier;
14. private Task editTask;
15.
16. public TaskEditorPanel(TaskChangeObservable newNotifier){
17.    notifier = newNotifier;
18.    createGui();
19. }
20. public void createGui(){
21.     setLayout(new BorderLayout());
22.     editPanel = new JPanel();
23.    editPanel.setLayout(new GridLayout(3, 2));
24.     taskName = new JTextField(20);
25.     taskNotes = new JTextField(20);
26.     taskTime = new JTextField(20);
27.    editPanel.add(new JLabel("Task Name"));
28.     editPanel.add(taskName);
29.    editPanel.add(new JLabelC'Task Notes"));
30.     editPanel.add(taskNotes);
31.     editPanel.add(new JLabel("Time Required"));
32.     editPanel.add(taskTime);
33.
34.     controlPanel = new JPanel();
35.     add = new JButton("Add Task");
36.    update = new JButton("Update Task");
37.     exit = new JButton("Exit");
38.    controlPanel.add(add);
39.     controlPanel.add(update);
40.    controlPanel.add(exit);
41.    add.addActionListener(this);
42.     update.addActionListener(this);
43.     exit.addActionListener(this);
44.     add(controlPanel, BorderLayout.SOUTH);
45.     add(editPanel, BorderLayout.CENTER);
46.  }
47.  public void setTaskChangeObservable(TaskChangeObservable newNotifier){
48.     notifier = newNotifier;
49.  }
50.  public void actionPerformed(ActionEvent event)(
51.     Object source = event.getSource();
52.     if (source == add){
53.       double timeRequired = 0.0;
54.       try{
55.            timeRequired = Double.parseDouble(taskTime.getText());
56.       }
57.       catch (NumberFormatException exc){}
58.     notifier.addTask(new Task(taskName.getText(), taskNotes.getText(),
   timeRequired));
59.     }
60.     else if (source == update){
61.       editTask.setName(taskName.getText());
62.       editTask.setNotes(taskNotes.getText());
63.       try{
64.            editTask.setTimeRequired(Double.parseDouble(taskTime.getText()));
65.       }
66.       catch (NumberFormatException exc){}
67.      notifier.updateTask(editTask);
68.     }
69.     else if (source == exit){
70.      System.exit(0);
71.     }
72.
```

```
73. }
74. public void task taskAdded(Task task){ }
75. public void taskChanged(Task task){ )
76. public void taskSelected(Task task){
77.    editTask = task;
78.  taskName.setText (task.getName() ) ;
79.   taskNotes.setText(task.getNotes() ) ;
80.   taskTime.setText("" + task.getTimeRequired() );
81. )
82.}
```

Листинг A.93. TaskHistoryPanel.java

```
1. import java.awt.BorderLayout;
2. import javax.swing.JPanel;
3. import javax.swing.JScrollPane;
4. import javax.swing.JTextArea;
5. public class TaskHistoryPanel extends JPanel implements TaskChangeObserver{
6.  private JTextArea displayRegion;
7.
8.  public TaskHistoryPanel(){
9.    createGui () ;
10. }
11. public void createGui (){
12.    setLayout(new BorderLayout());
13.   displayRegion = new JTextArea(10, 40);
14.    displayRegion.setEditable(false);
15.    add(new JScrollPane(displayRegion));
16. }
17. public void taskAdded(Task task)(
18.   displayRegion.append("Created task" + task + "\n");
19. )
20. public void taskChanged(Task task){
21.   displayRegion.append("Updated task " + task + "\n");
22. }
23. public void taskSelected(Task task){
24.   displayRegion.append("Selected task " + task + "\n");
25. }
26.}
```

Листинг A.94. TaskSelectorPanel.java

```
1. import java.awt.event.ActionEvent;
2. import java.awt.event.ActionListener;
3. import javax.swing.JPanel;
4. import javax.swing.JComboBox;
5. public class TaskSelectorPanel extends JPanel implements ActionListener,
   TaskChangeObserver{
6.  private JComboBox selector = new JComboBox();
7.  private TaskChangeObservable notifier;
8.  public TaskSelectorPanel(TaskChangeObservable newNotifier){
9.    notifier = newNotifier;
10.    createGui();
11. }
12. public void createGui (){
13.    selector = new JComboBox ();
14.   selector.addActionListener(this) ;
15.   add(selector) ;
```

```
16. }
17. public void actionPerformed(ActionEvent evt){
18.   notifier.selectTask((Task)selector.getSelectedItem()) ;
19. }
20. public void setTaskChangeObservable(TaskChangeObservable newNotifier) {
21.   notifier = newNotifier;
22. }
23.
24. public void taskAdded(Task task){
25.   selector.addItem(task);
26. }
27. public void taskChanged(Task task){ }
28. public void taskSelected(Task task){ }
29.}
```

Одно из свойств, присущих шаблону Observer, состоит в том, что класс наблюдаемого объекта Observable использует стандартный интерфейс для своих наблюдателей (в данном случае он называется TaskChangeObserver). Это означает не только то, что шаблон Observer является более универсальным, чем, скажем, шаблон Mediator, но и то, что объекты-наблюдатели могут получать определенное количество ненужных им сообщений. Например, класс TaskEditorPanel, как это видно из листинга А.92, не выполняет никаких действий при вызове его методов taskAdded и taskChanged.

Класс Task (листинг А.95) представляет в пользовательском интерфейсе прикладной объект, который в данном примере моделирует определенный этап проекта.

Листинг А.95. Task.java

```
1. public class Task{
2.   private String name = "";
3.   private String notes = "";
4.   private double timeRequired;
5.
6.   public Task(){ }
7.   public Task(String newName, String newNotes, double newTimeRequired) {
8.     name = newName;
9.     notes = newNotes;
10.    timeRequired = newTimeRequired;
11. }
12.
13.  public String getName(){ return name; }
14.  public String getNotes(){ return notes; }
15.  public double getTimeRequired(){ return timeRequired; }
16.  public void setName(String newName); name = newName; }
17.  public void setTimeRequired(double newTimeRequired){ timeRequired =
   newTimeRequired;}
18.  public void setNotes(String newNotes)( notes = newNotes; }
19.  public String toStringO { return name + " " + notes; }
20.}
```

За создание пользовательского интерфейса в данном примере отвечает класс RunPattern (листинг А.96), который создает наблюдаемый объект и объекты-наблюдатели.

Листинг A.96. RunPattern.java

```
1. public class RunPattern{
2.    public static void main(String [] arguments){
3.       System.out.println("Example for the Observer pattern");
4.       System.out.println("This demonstration uses a central observable");
5.       System.out.println(" object to send change notifications to several");
6.       System.out.println(" JPanels in a GUI. Each JPanel is an Observer,");
7.       System.out.println(" receiving notifications when there has been some");
8.       System.out.println(" change in the shared Task that is being edited.");
9.       System.out.println();
10.
11.      System.out.println("Creating the ObserverGui");
12.      ObserverGui application = new ObserverGui ();
13.      application.createGui();
14.   }
15.}
```

State

Объекты States лучше всего определять в виде внутренних классов, так как это обеспечивает им тесную связь с основным классом, а также прямой доступ к атрибутам последнего. В приведенном ниже примере показано, как это работает на практике.

Стандартный подход к сохранению информации в приложениях состоит в том, чтобы выполнять такие операции только по мере необходимости, т.е. после внесения изменений. Когда изменения внесены, но файл не сохранен, его состояние называют "плохим" (dirty). Это означает, что содержимое открытого файла может отличаться от содержимого ранее сохраненной версии. Файл, который был сохранен и после сохранения которого не вносились никакие изменения, называют "хорошим" (clean). В "хорошем" состоянии содержимое открытого и сохраненного файлов идентично.

В данном примере показано, как использовать шаблон State для обновления объектов Appointment PIM-приложения с сохранением их, по мере необходимости, в файле. Диаграмма перехода состояний для файла представлена на рис. A. 1.

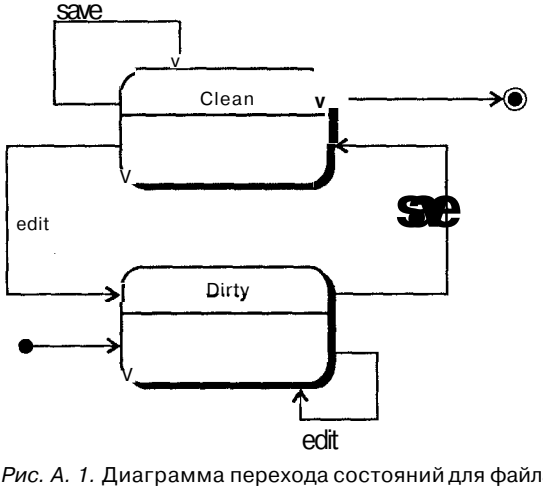


Рис. A. 1. Диаграмма перехода состояний для файла

Два класса состояний (CleanState и DirtyState) реализуют интерфейс State (листинг А.97). Эти классы отвечают за определение следующего состояния (в данном случае это несложно, так как имеется всего лишь два состояния).

В интерфейсе State определены два метода— save и edit, которые в случае необходимости вызываются классом CalendarEditor (листинг А.98).

Листинг А.97. State.java

```
1. public interface State{
2.   public void saved;
3.   public void edit();
4. )
```

Класс CalendarEditor управляет коллекцией объектов Appointment.

Листинг А.98. CalendarEditor.java

```
1. import java.io.File;
2. import java.util.ArrayList;
3. public class CalendarEditor{
4.   private State currentState;
5.   private File appointmentFile;
6.   private ArrayList appointments = new ArrayList();
7.   private static final String DEFAULT_APPOINTMENT_FILE = "appointments.ser";
8.
9.   public CalendarEditor(){
10   .  this(DEFAULT_APPOINTMENT_FILE);
11. }
12.  public CalendarEditor(String appointmentFileName){
13.    appointmentFile = new File (appointmentFileName);
14.    try{
15.      appointments =  (ArrayList)FileLoader.loadData(appointmentFile);
16.    }
17.    catch (ClassCastException exc){
18.      System.err.println("Unable to load information. The file does not
   contain a list of appointments.");
19.    }
20.    currentState = new CleanState();
21.)
22.
23.  public void save(){
24.    currentState.save();
25. )
26.
27.  public void edit(){
28.    currentState.edit();
29. }
30.
31.  private class DirtyState implements State{
32.    private State nextState;
33.
34.    public DirtyState(State nextState){
35.      this.nextState = nextState
36.    }
37.
38.    public void save Of
39.      FileLoader.storeData(appointmentFile, appointments);
40.      currentState = nextState;
```

```
41.    }
42.    public void edit(){ }
43. }
44.
45. private class CleanState implements State{
46.    private State nextState = new DirtyState(this);
47.
48.    public void saved { }
49.    public void edit(){ currentState = nextState; }
50. }
51.
52. public ArrayList getAppointments(){
53.    return appointments;
54. }
55.
56. public void addAppointment(Appointment appointment){
57.    if (!appointments.contains(appointment)){
58.       appointments.add(appointment) ;
59.    }
60. }
61. public void removeAppointment(Appointment appointment){
62.    appointments.remove(appointment);
63. }
64.}
```

Класс StateGui (листинг А.99) обеспечивает интерфейс редактирования для событий, запланированных с помощью редактора CalendarEditor. Следует заметить, что объект GUI хранит ссылку на CalendarEditor, с помощью которой делегирует редактору права на внесение изменений или сохранение. Это позволяет редактору выполнять требуемые операции и обновлять в случае необходимости свое состояние.

Листинг A.99. StateGui.java

```
1. import java.awt.Container;
2. import java.awt.BorderLayout;
3. import java.awt.event.ActionListener;
4. import java.awt.event.WindowAdapter;
5. import java.awt.event.ActionEvent;
6. import java.awt.event.WindowEvent;
7. import javax.swing.BoxLayout;
8. import javax.swing.JButton;
9. import javax.swing.JComponent;
10.import javax.swing.JFrame;
11.import javax.swing.JPanel;
12.import javax.swing.JScrollPane;
13.import javax.swing.JTable;
14.import javax.swing.table.AbstractTableModel;
15.import java.util.Date;
16.public class StateGui implements ActionListener{
17.  private JFrame mainFrame;
18.  private JPanel controlPanel, editPanel;
19.  private CalendarEditor editor;
20.  private JButton save, exit;
21.
22.  public StateGui(CalendarEditor edit){
23.     editor = edit;
24.  }
25.
26.  public void createGui(){
```

```
27.    mainFrame = new JFrame("State Pattern Example");
28.    Container content = mainFrame.getContentPane ();
29.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
30.
31.    editPanel = new JPanel();
32.    editPanel.setLayout(new BorderLayout());
33.    JTable appointmentTable = new JTable(new StateTableModel((Appointment
    []editor.getAppointments().toArray(newAppointment[1])));
34.    editPanel.add(new JScrollPane(appointmentTable));
35.    content.add(editPanel);
36.
37.    controlPanel = new JPanel();
38.    save = new JButton("Save Appointments");
39.    exit = new JButton("Exit");
40.    controlPanel.add(save);
41.    controlPanel.add(exit);
42.    content.add(controlPanel);
43.
44.    save.addActionListener(this);
45.    exit.addActionListener(this);
46.
47.    mainFrame.addWindowListener(new WindowCloseManager());
48.    mainFrame.pack();
49.    mainFrame.setVisible(true);
50. }
51.
52.
53. public void actionPerformed(ActionEvent evt){
54.    Object originator = evt.getSource();
55.    if (originator == save){
56.      saveAppointments();
57.    }
58.    else if (originator == exit){
59.     exitApplication() ;
60.    }
61. }
62.
63. private class WindowCloseManager extends WindowAdapter{
64.    public void windowClosing(WindowEvent evt){
65.      exitApplication();
66.    )
67. }
68.
69. private void saveAppointments(){
70.  editor. save () ;
71. }
72.
73. private void exitApplication(){
74.    System.exit(0);
75. }
76.
77. private class StateTableModel extends AbstractTableModel{
78.    private final String [] columnNames = (
79.      "Appointment", "Contacts", "Location", "Start Date", "End Date" };
80.    private Appointment [] data;
81.
82.    public StateTableModel(Appointment [] appointments){
83.      data = appointments;
84.    }
85.
86.    public String getColumnName(int column)(
87.      return columnNames[column] ;
88.    }
```

```
89.     public int getRowCount(){ return data.length; }
90.     public int getColumnCount(){ return columnNames.length; }
91.     public Object getValueAT(int row, int column))
92.       Object value = null;
93.       switch(column){
94.            case 0: value = data[row].getReason();
95.                 break;
96.            case 1: value = data[row].getContacts();
97.                 break;
98.            case 2: value = data[row].getLocation();
99.                 break;
100.           case 2: value = data[row].getStartDate();
101.                break;
102.           case 4: value = data[row].getEndDate();
103.           break;
104.      }
105.      return value;
106.    }
107.    public boolean isCellEditable(int row, int column){
108.      return ((column ==0) || (column ==2)) ? true : false;
109.    }
110.    public void setValueAt(Object value, int row, int column)(
111.      switch(column){
112.           case 0: data[row].setReason((String)value);
113.                editor.edit();
114.                break;
115.           case 1:
116.                break;
117.           case 2: data[row].setLocation(new LocationImpl((String)value));
118.                editor.edit();
119.                break;
120.           case 3:
121.                break;
122.           case 4:
123.                break;
124.      }
125.    }
126. }
127.}
```

В данном примере используется пять вспомогательных прикладных классов и интерфейсов: Appointment (листинг А.100), Contact (листинг А.101), ContactImpl (листинг А.102), Location (листинг А. 103) и LocationImpl (листинг А. 104).

Листинг А.100. Appointment.java

```
1. import java.io.Serializable;
2. import java.util.Date;
3. import java.util.ArrayList;
4. public class Appointment implements Serializable{
5.   private String reason;
6.   private ArrayList contacts;
7.   private Location location;
8.   private Date startDate;
9.   private Date endDate;
10.
11.  public Appointment(String reason, ArrayList contacts, Location location,
   Date startDate, Date endDate){
12.    this.reason = reason;
13.    this.contacts = contacts;
```

```
14.    this.location = location;
15.    this.startDate = startDate;
16.    this.endDate = endDate;
17. }
18.
19. public String getReason(){ return reason; }
20. public ArrayList getContacts(){ return contacts; }
21. public Location getLocation(){ return location; }
22. public Date getStartDate(){ return startDate; }
23. public Date getEndDate(){ return endDate; }
24.
25. public void setReason(String reason){ this.reason = reason; }
26. public void setContacts(ArrayList contacts){ this.contacts = contacts; }
27. public void setLocation(Location location){ this.location = location; }
28. public void setStartDate(Date startDate){ this.startDate = startDate; }
29. public void setEndDate(Date endDate){ this.endDate = endDate; }
30.
31. public String toString(){
32.   return "Appointment:" + "\n Reason: " + reason +
33.     "\n Location: " + location + "\n Start: " +
34.     startDate + "\n End: " + endDate + "\n";
35. }
36. }
```

Листинг А. 101. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization) ;
13.}
```

Листинг А. 102. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14.    }
15.
16. public String getFirstName(){ return firstName; }
```

```
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization;}
25.
26. public String toString(){
27.   return firstName + SPACE + lastName;
28. }
29.}
```

Листинг А. 103. Location.java

```
1. import java.io.Serializable;
2. public interface Location extends Serializable{
3.   public String getLocation ();
4.   public void setLocation(String newLocation);
5. }
```

Листинг А. 104. LocationImpl.java

```
1. public class LocationImpl implements Location{
2.   private String location;
3.
4.   public LocationImpl() ( )
5.   public LocationImpl(String newLocation){
6.      location = newLocation;
7.   }
8.
9.   public String getLocation(){ return location; }
10.
11.  public void setLocation(String newLocation){ location = newLocation; }
12.
13.  public String toString(){ return location; }
14.}
```

Для создания простого множества объектов класса Appointment используется класс DataCreator (листинг А.105), а для управления их сохранением в файле и извлечением из файла — класс FileLoader (листинг А. 106).

Листинг А. 105. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. import java.util.Calendar;
6. import java.util.Date;
7. import java.util.ArrayList;
```

```
8. public class DataCreator{
9.   private static final String DEFAULT_FILE = "data.ser";
10.  private static Calendar dateCreator = Calendar.getInstance();
11.
12.  public static void main(String [] args){
13.    String fileName;
14.    if (args.length == 1) {
15.      fileName = args[0];
16.    }
17.    else{
18.      fileName = DEFAULT_FILE;
19.    }
20.    serialize(fileName);
21. }
22.
23.  public static void serialize(String fileName){
24.    try(
25.    serializeToFile(createData() , fileName);
26.    )
27.    catch (IOException exc) {
28.      exc.printStackTrace();
29.    }
30. }
31.
32.  private static Serializable createData(){
33.    ArrayList appointments = new ArrayList ();
34.      ArrayList contacts = new ArrayList();
35.      contacts.add(new ContactImpl("Test", "Subject", "Volunteer", "United
   Patterns Consortium"));
36.      Location location1 = new LocationImpl("Punxsutawney, PA");
37.      appointments.add(new Appointment("Slowpokes anonymous", contacts,
   location1, createDate(2001, 1, 1, 12, 01), createDate(2001, 1, 1, 12, 02)));
38.      appointments.add(new Appointment("Java focus group", contacts,
   location1, createDate(2001, 1, 1, 12, 30), createDate(2001, 1, 1, 14, 30)));
39.      appointments.add(new Appointment("Something else", contacts, location1,
   createDate(2001, 1, 1, 12, 01), createDate(2001, 1, 1, 12, 02)));
40.      appointments.add(new Appointment("Yet another thingie", contacts,
   location1, createDate(2001, 1, 1, 12, 01), createDate(2001, 1, 1, 12, 02)));
41.      return appointments;
42.  }
43.
44.  private static void serializeToFile(Serializable content, String fileName)
   throws IOException{
45.      ObjectOutputStream serOut = new ObjectOutputStream(new
   FileOutputStream(fileName));
46.      serOut.writeObject(content);
47.      serOut.close();
48.  }
49.
50.  public static Date createDate(int year, int month, int day, int hour, int
   minute){
51.     dateCreator.set(year, month, day, hour, minute);
52.      return dateCreator.getTime();
53.  }
54.}
```

Листинг A. 106. FileLoader.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. import java.io.ObjectInputStream;
6. import java.io.ObjectOutputStream;
7. import java.io.Serializable;
8. public class FileLoader{
9.  public static Object loadData(File inputFile){
10.    Object returnValue = null;
11.    try{
12.     if (inputFile.exists() ){
13.          if (inputFile.isFile()){
14.              ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(inputFile));
15.                  returnValue = readIn.readObject();
16.                  readIn.close ();
17.          }
18.         else f
19.            System.err.println(inputFile + " is a directory.");
20.          }
21.      )
22.      else {
23.         System.err.println("File " + inputFile + " does not exist.");
24.      ;
25.    }
26.    catch (ClassNotFoundException exc){
27.      exc.printStackTrace();
28.
29.    }
30.    catch (IOException exc){
31.      exc.printStackTrace();
32.
33.    }
34.    return returnValue;
35. }
36.  public static void storeData(File outputFile, Serializable data){
37.    try{
38.     ObjectOutputStream writeOut = new ObjectOutputStream(new
   FileOutputStream(outputFile));
39.      writeOut.writeObject(data);
40.      writeOut.close();
41.    }
42.    catch (IOException exc){
43.     exc.printStackTrace ();
44.    }
45. }
46.}
```

Класс RunPattern (листинг A. 107) осуществляет запуск примера, создавая объект класса CalendarEditor (этот класс инициализируется начальными значениями запланированных событий, извлекаемых из файла) и отображая его содержимое с помощью объекта класса StateGui.

Листинг А.107. RunPattern.java

```
1. import java.io.File;
2. public class RunPattern{
3.   public static void main(String [] arguments){
4.       System.out.println("Example for the State pattern");
5.       System.out.println();
6.
7.       if (!(new File("appointments.ser").exists())){
8.         DataCreator.serialize("appointments.ser");
9.       }
10.
11.      System.out.println("Creating CalendarEditor");
12.      CalendarEditor appointmentBook = new CalendarEditor();
13.      System.out.println("");
14.
15.      System.out.println("Created. Appointments:");
16.      System.out.println(appointmentBook.getAppointments());
17.
18.      System.out.println("Created. Creating GUI:");
19.      StateGui application = new StateGui(appointmentBook);
20.      application.createGui();
21.      System.out.println("");
22. )
23.}
```

Strategy

Многие коллекции PIM-приложения могут использовать определенные способы организации хранящихся в них элементов и получения сводной информации о них. В данном примере показано, как с помощью шаблона Strategy можно обеспечить получение сводной информации об элементах коллекции ContactList (листинг А.108), которые представляют собой объекты класса Contact.

Листинг А.108. ContactList.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public class ContactList implements Serializable{
4.  private ArrayList contacts = new ArrayList ();
5.  private SummarizingStrategy summarizer;
6.
7.  public ArrayList getContacts(){ return contacts; }
8.  public Contact [] getContactsAsArray(){ return (Contact []
   (contacts.toArray(newContact [1])); }
9.
10. public void setSummarizer(SummarizingStrategy newSummarizer){ summarizer =
   newSummarizer; }
11. public void setContacts(ArrayList newContacts){ contacts = newContacts; }
12.
13. public void addContact(Contact element){
14.    if (!contacts.contains(element)){
15.      contacts.add(element);
16.    }
17. }
18. public void removeContact(Contact element){
```

```
19.    contacts.remove(element) ;
20. }
21.
22.  public String summarize(){
23.     return summarizer.summarize(getContactsAsArray());
24. }
25.
26.  public String [] makeSummarizedList(){
27.     return summarizer.makeSummarizedList(getContactsAsArray());
28. }
29.}
```

Класс ContactList имеет два метода, с помощью которых коллекция может получать сводную информацию о хранящихся в ней объектах Contact— summarize и makeSummarizedList. Оба метода делегированы интерфейсу SummarizingStrategy (листинг А.109), ссылка на экземпляр которого устанавливается методом setSummarizer класса коллекции.

Листинг А.109. SummarizingStrategy.java

```
1. public interface SummarizingStrategy{
2.   public static final String EOL_STRING =
   System.getProperty("line.separator");
3.   public static final String DELIMITER = ":";
4.   public static final String COMMA = ",";
5.   public static final String SPACE = " ";
6.
7.   public String summarize(Contact [] contactList);
8.   public String [] makeSummarizedList(Contact [] contactList);
9. }
```

SummarizingStrategy— это интерфейс, который определяет два делегированных ему метода summarize и makeSummarizedList. Данный интерфейс соответствует интерфейсу Strategy в иерархии классов шаблона Strategy. В рассматриваемом примере класс ConcreteStrategy представлен двумя классами: NameSummarizer (листинг АЛЮ) и OrganizationSummarizer (листинг А.1 11). Оба класса обеспечивают получение сводной информации об элементах списка контактов, однако каждый из них представляет собственный набор информации и группирует данные по-своему.

Класс NameSummarizer возвращает лишь имена и фамилии контактных лиц, первой указывая фамилию. Сравнение выполняется внутренним классом NameComparator, который выполняет сортировку всех элементов списка в алфавитном порядке сначала по фамилиям, а затем по именам.

Листинг А.110. NameSummarizer.java

```
1. import java.text.Collator;
2. import java.util.Arrays;
3. import java.util.Comparator;
4. public class NameSummarizer implements SummarizingStrategy{
5.   private Comparator comparator = new NameComparator();
6.
7.   public String summarize(Contact [] contactList){
8.      StringBuffer product = new StringBuffer();
```

```
9.    Arrays.sort(contactList, comparator);
10.   for (int i = 0; i< contactList.length; i++){
11.   product.append(contactList[i] .getLastName() ) ;
12.   .   product.append(COMMA);
13.       product.append(SPACE);
14.  . product.append(contactList[i J .getFirstName());
15.       product.append(EOL_STRING);
16.    }
17.    return product.toString();
18. }
19.
20. public String[] makeSummarizedList(Contact [] contactList){
21.   Arrays.sort(contactList, comparator);
22.   String [] product = new String[contactList.length]
23.   for (int i = 0; i < contactList.length; i++){
24.    product[i] = contactList[i].getLastName() + COMMA + SPACE +
25.         contactList[i].getFirstName() + EOL_STRING;
26.   }
27.   return product;
28. >
29.
30. private class NameComparator implements Comparator{
31.   private Collator textComparator = Collator.getInstance();
32.
33.   public int compare(Object o1, Object o2){
34.     Contact c1, c2;
35.    if ( (ol instanceof Contact) && (o2 instanceof Contact)){
36.         c1 = (Contact)o1;
37.         c2 = (Contact)o2;
38.         int compareResult = textComparator.compare(c1.getLastName(),
    c2.getLastName());
39.         if (compareResult == 0)(
40.              compareResult = textComparator.compare(c1.getFirstName(),
    c2.getFirstName());
41.          }
42.          return compareResult;
43.     }
44.     else return textComparator.compare(o1,o2);
45.   }
46.
47.   public boolean equals(Object o){
48.     return textComparator.equals(o);
49.   }
50. }
51.}
```

Класс OrganizationSummarizer представляет контактную информацию, отсортированную по организациям с указанием имени и фамилии контактного лица. Объект класса Comparator обеспечивает сортировку информации в алфавитном порядке сначала по организациям, а затем по фамилиям.

Листинг А. 111. OrganizationSummarizer.java

```
1. import java.text.Collator;
2. import java.util.Arrays;
3. import java.util.Comparator;
4. public class OrganizationSummarizer implements SummarizingStrategy{
5.  private Comparator comparator = new OrganizationComparator()
6.
7.  public String summarize(Contact [] contactList){
```

```
8.    StringBuffer product = new StringBuffer()
9.    Arrays.sort(contactList, comparator);
10.   for (int i = 0; i < contactList.length; i++){
11.     product.append(contactList[i] .getOrganization());
12.      product.append(DELIMITER);
13.      product.append(SPACE);
14.      product.append(contactList[i].getFirstName());
15.      product.append(SPACE);
16.      product.append(contactList[i].getLastName());
17.      product.append(EOL_STRING);
18.    }
19.  return product.toString () ;
20. }
21.
22. public String [] makeSummarizedList(Contact [] contactList){
23.   Arrays.sort(ContactList, comparator);
24.   String [], product = new String[contactList.length];
25.   for (int i = 0; i < contactList.length; i++){
26.    product[i] = contactList[i].getOrganization() + DELIMITER + SPACE +
27.         contactList[i].getFirstName() + SPACE +
28.         contactList[i].getLastName() + EOL_STRING;
29.    }
30.   return product;
31. }
32.
33. private class OrganizationComparator implements Comparator{
34.   private Collator textComparator = Collator.getInstance();
35.
36.   public int compare(Object o1, Object o2){
37.     Contact c1,c2;
38.     if ( (o1 instanceof Contact) && (o2 instanceof Contact)){
39.          c1 = (Contact)o1;
40.          c2 = (Contact)o2;
41.          int compareResult = textComparator.compare(c1.getOrganization(),
   c2.getOrganization());
42.          if (compareResult == 0){
43.          compareResult = textComparator.compare(c1.getLastName(),
   c2.getLastName());
44.          }
45.          return compareResult;
46.      }
47.     else return textComparator.compare(o1, o2);
48.    }
49.
50.   public boolean equals(Object o){
51.     return textComparator.equals(o);
52.    }
53. }
54.}
```

Класс ContactList использует интерфейс Contact (листинг A.1 12) и реализующий его интерфейс ContactImpl (листинг A. 113) для представления отдельных контактов.

Листинг A.112. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
```

```
6.  public String getTitle();
7.  public String getOrganization ();
8.
9.  public void setFirstName(String newFirstName);
10. public void setLastName(String newLastName);
11. public void setTitle(String newTitle);
12. public void setOrganization(String newOrganization) ;
13.}
```

Листинг А. 113. ContactList.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization; }
25.
26. public String toString(){
27.    return firstName + SPACE + lastName;
28. }
29.}
```

Классы DataCreator (листинг А.1 14) и DataRetriever (листинг А. 115) применяются для сохранения группы контактов в файле и извлечения ее из файла.

Листинг А.114. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. public class DataCreator{
6.   private static final String DEFAULT_FILE = "data.ser";
7.
8.   public static void main(String [] args){
9.      String fileName;
```

```
10.     if (args.length == 1){
11.       fileName = args[0];
12.     }
13.     else{
14.       fileName = DEFAULT_FILE;
15.     }
16.     serialize(fileName);
17.  }
18.
19.  public static void serialize(String fileName){
20.     try{
21.       serializeToFile(makeContactList(), fileName);
22.     }
23.     catch (IOException exc){
24.       exc.printStackTrace();
25.     }
26.  }
27.
28.  private static Serializable makeContactList(){
29.     ContactList list = new ContactList();
30.     list.addContact(new ContactImpl("David", "St. Hubbins", "Lead Guitar",
    "The New Originals"));
31.     list.addContact(new ContactImpl("Mick", "Shrimpton", "Drummer", "The New
    Originals"));
32.     list.addContact(new ContactImpl("Nigel", "Tufnel", "Lead Guitar", "The
    New Originals"));
33.     list.addContact(new ContactImpl("Derek", "Smalls", "Bass", "The New
    Originals"));
34.     list.addContact(new ContactImpl("Viv", "Savage", "Keyboards", "The New
    Originals"));
35.     list.addContact(new ContactImpl("Nick", "Shrimpton", "CEO", "Fishy
    Business, LTD"));
36.     list.addContact(new ContactImpl("Nickolai", "Ludachevski", "Senior
    Packer", "Fishy Business, LTD"));
37.     list.addContact(new ContactImpl("Alan", "Robertson", "Comptroller",
    "Universal Exports"));
38.     list.addContact(new ContactImpl("William", "Telle", "President",
    "Universal Exports"));
39.     list.addContact(new ContactImpl("Harvey", "Manfredjensenden",
    "Inspector", "Universal Imports"));
40.     list.addContact(new ContactImpl("Deirdre", "Pine", "Chief Mechanic",
    "The Universal Joint"));
41.     list.addContact(new ContactImpl("Martha", "Crump-Pinnett", "Lead
    Developer", "Avatar Inc."));
42.     list.addContact(new ContactImpl("Bryan", "Basham", "CTO", "IOVA"));
43.     return list;
44.  }
45.
46.  private static void serializeToFile(Serializable content, String fileName)
    throws IOException{
47.     ObjectOutputStream serOut = new ObjectOutputStream(new
    FileOutputStream(fileName));
48.     serOut.writeObject(content);
49.     serOut.close();
50.  }
51.}
```

Листинг А. 115. DataRetriever.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5.
6. public class DataRetriever{
7.   public static Object deserializeData(String fileName){
8.      Object returnValue = null;
9.      try{
10.       File inputFile = new File(fileName) ;
11.       if (inputFile.exists() && inputFile.isFile()){
12.            ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(fileName));
13.            returnValue = readIn.readObject();
14.            readIn.close();
15.       }
16.       else {
17.            System.err.println("Unable to locate the file " + fileName);
18.       }
19.     }
20.     catch (ClassNotFoundException exc) {
21.       exc.printStackTrace() ;
22.
23.     }
24.     catch (IOException exc) {
25.       exc.printStackTrace();
26.
27.     }
28.     return returnValue;
29. }
30.}
```

Класс RunPattern (листинг А. 116) демонстрирует работу примера для рассматриваемого шаблона. Этот класс сначала создает список контактов, а затем выводит содержащиеся в нем элементы, используя для этого оба объекта класса SummarizingStrategy.

Листинг А. 116. RunPattern. java

```
1. import java.io.File;
2. public class RunPattern{
3.   public static void main(String [] arguments){
4.      System.out.println("Example for the Strategy pattern");
5.      System.out.println() ;
6.      System.out.println("This code uses two Strategy classes, NameSummarizer
   and OrganizationSummarizer,");
7.      System.out.println(" to provide a sorted, summarized list for a
   ContactList. The ContactList object");
8.      System.out.println(" maintains a collection of Contacts, and delegates
   the task of representing");
9.      System.out.println(" its information to an associated object which
   implements SummarizingStrategy.");
10.     System.out.println ();
11.
12.     System.out.println("Deserializing stored ContactList from the data.ser
   file");
13.     System.out.println() ;
```

```
14.    if (!(new File("data.ser") .exists())){
15.      DataCreator.serialize("data.ser");
16.    }
17.    ContactList list =
   (ContactList)(DataRetriever.deserializeData("data.ser"));
18.
19.    System.out.println("Creating NameSummarizer for the ContactList");
20.    System.out.println("(this Strategy displays only the last and first name,");
21.    System.out.println(" and sorts the list by last name, followed by the
   first)");
22.    list.setSummarizer(new NameSummarizer());
23.
24.    System.out.println("Name Summarizer Output for the ContactList:");
25  .  System.out.println(list.summarize());
26.    System.out.println();
27.
28.    System.out.println("Creating OrganizationSummarizer for the ContactList");
29.    System.out.println("(this Strategy displays the organization, followed
   by the first");
30.    System.out.println(" and last name. It sorts by the organization,
   followed by last name)");
31.    list.setSummarizer(new OrganizationSummarizer());
32.
33.    System.out.println ("Organization Summarizer Output for the ContactList:");
34 . System.out.println(list.summarize() ) ;
35.    System.out.println();
36. }
37.}
```

Visitor

Шаблон Visitor часто используется в тех случаях, когда по ходу операций, выполняемых над структурами большого объема, необходимо вычислять промежуточные и итоговые результаты. В данном примере показано, как с помощью шаблона Visitor обеспечивается выполнение расчета общей стоимости проекта.

Для представления элементов проекта используется четыре класса, каждый из которых представляет собой реализацию общего для всех классов интерфейса ProjectItem (листинг A. 117). В данном примере этот интерфейс содержит определение метода accept, предназначенного для сохранения ссылки на экземпляр класса Visitor.

Листинг A.117. ProjectItem.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public interface ProjectItem extends Serializable{
4.   public void accept(ProjectVisitor v);
5.   public ArrayList getProjectItems();
6. }
```

Собственно проект представлен классом Project (листинг A.120). Класс Deliverable (листинг A.118) представляет конкретный продукт проекта, а класс Task (листинг A.121) — его отдельный этап. Кроме того, в листинге A.119 представлен подкласс класса Task, названный DependentTask. Этот класс содержит набор ссылок на объекты класса Task, влияющие на завершение этапа проекта, который представлен конкретным экземпляром класса DependentTask.

Листинг А.118. Deliverable.java

```
1. import java.util.ArrayList;
2. public class Deliverable implements ProjectItem{
3.   private String name;
4.   private String description;
5.   private Contact owner;
6.   private double materialsCost;
7.   private double productionCost;
8.
9.   public Deliverable (){ }
10.  public Deliverable(String newName, String newDescription,
11.     Contact newOwner, double newMaterialsCost, double newProductionCost){
12.     name = newName;
13.     description = newDescription;
14.     owner = newOwner;
15.     materialsCost = newMaterialsCost;
16.     productionCost = newProductionCost;
17. }
18.
19.  public String getName(){ return name; }
20.  public String getDescription (){ return description; }
21.  public Contact getOwner(){ return owner; }
22.  public double getMaterialsCost(){ return materialsCost; }
23.  public double getProductionCost (){ return productionCost; }
24.
25.  public void setMaterialsCost(double newCost){ materialsCost = newCost; }
26.  public void setProductionCost(double newCost){ productionCost = newCost; }
27.  public void setName(String newName){ name = newName: }
28.  public void setDescription(String newDescription){ description =
   newDescription; }
29.  public void setOwner(Contact newOwner){ owner = newOwner; }
30.
31.  public void accept(ProjectVisitor v){
32.     v.visitDeliverable(this);
33. }
34.
35.  public ArrayList getProjectItems(){
36.     return null;
37. }
38.}
```

Листинг А.119. DependentTask.java

```
1. import java.util.ArrayList;
2. public class DependentTask extends Task{
3.   private ArrayList dependentTasks = new ArrayList ();
4.   private double dependencyWeightingFactor;
5.
6.   public DependentTask(){ }
7.   public DependentTask(String newName, Contact newOwner,
8.      double newTimeRequired, double newWeightingFactor){
9.      super(newName, newOwner, newTimeRequired);
10.     dependencyWeightingFactor = newWeightingFactor;
11.  }
12.
13.  public ArrayList getDependentTasks(){ return dependentTasks; }
14.  public double getDependencyWeightingFactor(){ return
   dependencyWeightingFactor; )
15.
```

```
16. public void setDependencyWeightingFactor(double newFactor){
  dependencyWeightingFactor = newFactor; }
17.
18. public void addDependentTask(Task element){
19.    if (!dependentTasks.contains(element)){
20.      dependentTasks.add(element);
21.    }
22. }
23.
24. public void removeDependentTask(Task element){
25.    dependentTasks.remove(element);
26. }
27.
28.  public void accept(ProjectVisitor v){
29.     v.visitDependentTask(this);
30.  }
31.}
```

Листинг A.120. Project.java

```
1. import java.util.ArrayList;
2. public class Project implements ProjectItem{
3.   private String name;
4.   private String description;
5.   private ArrayList projectItems = new ArrayList();
6.
7.  public Project)) { }
8.   public Project(String newName, String newDescription){
9.      name = newName;
10.     description = newDescription;
11.  }
12.
13.  public String getName(){ return name; }
14.  public String getDescription(){ return description; }
15.  public ArrayList getProjectItems(){ return projectItems; }
16.
17.  public void setName(Sting newName){ name = newName; }
18.  public void setDescription(String newDescription){ description =
   newDescription; }
19.
20.  public void addProjectItem(ProjectItem element){
21.     if (!projectItems.contains(element)){
22.       projectItems.add(element);
23.     }
24.  }
25.
26.  public void removeProjectItem(ProjectItem element){
27.     projectItems.remove(element);
28.  }
29.
30.  public void accept(ProjectVisitor v){
31.     v.visitProject(this);
32.  }
33.}
```

Листинг А.121. Task.java

```
1. import java.util.ArrayList;
2. public class Task implements ProjectItem{
3.   private String name;
4.   private ArrayList projectItems = new ArrayList ();
5.   private Contact owner;
6.   private double timeRequired;
7.
8.   public TaskO { }
9.   public Task(String newName, Contact newOwner,
10.     double newTimeRequired){
11.     name = newName;
12.     owner = newOwner;
13.     timeRequired = newTimeRequired;
14.  }
15.
16.  public String getName(){ return name; }
17.  public ArrayList getProjectItems(){ return projectltems; }
18.  public Contact getOwner(){ return owner; }
19.  public double getTimeRequired(){ return timeRequired; }
20.
21.  public void setName(String newName)( name = newName; }
22.  public void setOwner(Contact newOwner){ owner = newOwner; }
23.  public void setTimeRequired(double newTimeRequired){ timeRequired =
   newTimeRequired; )
24.
25.  public void addProjectItem(ProjectItem element){
26.  if (!projectItems.contains(element)){
27.     projectItems.add(element);
28.  )
29.  }
30.
31.  public void removeProjectItem(ProjectItem element){
32.     projectItems.remove(element);
33.  }
34.
35.  public void accept(ProjectVisitor v){
36.     v.visitTask (this);
37.  }
38.}
```

Базовый интерфейс ProjectVisitor, определяющий поведение классов вида Visitor, представлен в листинге А.122. Для каждого класса проекта интерфейс определяет соответствующий метод vis it.

Листинг А.122. ProjectVisitor.java

```
1. public interface ProjectVisitor{
2.   public void visitDependentTask(DependentTask p) ;
3.   public void visitDeliverable(Deliverable p) ;
4.   public void visitTask(Task p) ;
5.   public void visitProject(Project p) ;
6. }
```

Теперь, определив все основные классы, можно определить тот из них, который, реализует интерфейс ProjectVisitor и выполняет вычисления над элементами проекта. В листинге А.123 представлен такой класс, ProjectCostVisitor, предназначенный для расчета стоимости проекта.

Листинг А.123. ProjectCostVisitor.java

```
1. public class ProjectCostVisitor implements ProjectVisitor{
2.   private double totalCost;
3.   private double hourlyRate;
4.
5.   public double getHourlyRate(){ return hourlyRate; }
6.   public double getTotalCost(){ return totalCost; }
7.
8.   public void setHourlyRate(double rate){ hourlyRate = rate; }
9.
10.  public void resetTotalCost(){ totalCost = 0.0; }
11.
12.  public void visitDependentTask(DependentTask p){
13.    double taskCost = p.getTimeRequired() * hourlyRate;
14.    taskCost *= p.getDependencyWeightingFactor();
15.    totalCost += taskCost;
16.  }
17.  public void visitDeliverable(deliverable p){
18.    totalCost += p.getMaterialsCost() + p.getProductionCost();
19.  }
20.  public void visitTask(Task p){
21.    totalCost += p.getTimeRequired() * hourlyRate;
22.  }
23.  public void visitProject(Project p){ }
24.}
```

Вся функциональность по выполнению вычислений, равно как и хранение результатов централизованы в классе Visitor. Для того чтобы добавить поддержку новой функциональности, необходимо создать соответствующий класс, реализующий интерфейс ProjectVisitor, и наполнить нужным содержанием четыре метода visit этого класса.

Интерфейс Contact (листинг А. 124) и класс ContactImpl (листинг А.125) представляют владельца этапа проекта или продукта.

Листинг А.124. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Seriaiizable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А. 125. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl (){}
8.   public ContactImpl(String newFirstName, String newLastName/ String
   newTitle, String newOrganization){
9.        firstName = newFirstName;
10.       lastName = newLastName;
11.       title = newTitle;
12.       organization = newOrganization;
13. }
14.
15.  public String getFirstName(){ return firstName; }
16.  public String getLastName(){ return lastName; }
17.  public String getTitle(){ return title; }
18.  public String getOrganization(){ return organization; }
19.
20.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
21.  public void setLastName(String newLastName)( lastName = newLastName; }
22.  public void setTitle(String newTitle)( title = newTitle; )
23.  public void setOrganization(String newOrganization){ organization =
   newOrganization;}
24.
25.  public String toString()(
26.    return firstName + SPACE + lastName;
27. }
28.)
```

Представленный в листинге А. 126 вспомогательный класс DataCreator генерирует тестовый проект и сохраняет его в файле. Класс DataRetriever (листинг А.127) загружает данные проекта из файла, восстанавливая таким образом сохраненный проект.

Листинг А.126. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. public class DataCreator{
6.   private static final String DEFAULT_FILE = "data.ser";
7.
8.   public static void main(String [] args){
9.      String fileName;
10.     if (args.length == 1){
11.       fileName = args[0];
12.     )
13.     else{
14.       fileName = DEFAULT_FILE;
15.     }
16.     serialize(fileName);
17. }
18.
19.  public static void serialize(String fileName){
20.    try{
```

```
21.      serializeToFile(createData(), fileName);
22.     )
23.    catch (IOException exc) {
24.      exc.printStackTrace() ;
25.     }
26. }
27.
28. private static Serializable createData(){
29.   Contact contact = new ContactImpl("Test", "Subject", "Volunteer",
    "United Patterns Consortium");
30.
31.    Project project = new Project("Project 1", "Test Project");
32.
33.    Task task1 = new Task("Task 1", contact, 1);
34.    Task task2 = new Task("Task 2", contact, 1);
35.
36.    project.addProjectItem(new Deliverable("Deliverable 1", "Layer 1
    deliverable", contact, 50.0, 50.0));
37.    project.addProjectItem(task1);
38.    project.addProjectItem(task2);
39.    project.addProjectItem(new DependentTask("Dependent Task 1", contact, 1, 1));
40.
41.    Task task3 = new Task("Task 3", contact, 1) ;
42.    Task task4 = new Task("Task 4", contact, 1) ;
43.    Task task5 = new Task("Task 5", contact, 1) ;
44.    Task task6 = new Task("Task 6", contact, 1);
45.
46.    DependentTask dtask2 = new DependentTask("Dependent Task 2", contact, 1, 1);
47.
48.      task1.addProjectItem(task3);
49.      task1.addProjectItem(task4);
50.      task1.addProjectItem(task5);
51.      task1.addProjectItem(dtask2);
52.
53.      dtask2.addDependentTask(task5);
54.      dtask2.addDependentTask(task6);
55.      dtask2.addProjectItem(new Deliverable("Deliverable 2", "Layer 3
    deliverable", contact, 50.0, 50.0));
56.
57.      task3.addProjectItem(new Deliverable("Deliverable 3", "Layer 3
    deliverable", contact, 50.0, 50.0));
58.    task4.addProjectItem(new Task("Task 7", contact, 1) ) ;
59.      task4.addProjectItem(new Deliverable("Deliverable 4", "Layer 3
    deliverable", contact, 50.0, 50.0));
60.      return project;
61. )
62.
63. private static void serializeToFile(Serializable content, String fileName)
    throws IOException{
64.    ObjectOutputStream serOut = new ObjectOutputStream(new
    FileOutputStream(fileName));
65.    serOut.writeObject(content);
66.    serOut.close ();
67. }
68.}
```

Листинг А. 127. DataRetriever.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5.
6. public class DataRetriever{
7.   public static Object deserializeData(String fileName){
8.     Object returnValue = null;
9.     try{
10.      File inputFile = new File(fileName);
11.      if (inputFile.exists() && inputFile.isFile()){
12.          ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(fileName));
13.          returnValue = readIn.readObject();
14.          readIn.close();
15.      }
16.      else {
17.         System.err.println("Unable to locate the file " + fileName);
18.      }
19.    }
20.    catch (ClassNotFoundException exc){
21.      exc.printStackTrace();
22.
23.    }
24.    catch (IOException exc){
25.     exc.printStackTrace ();
26.
27.    }
28.    return returnValue;
29. }
30.}
```

Для того чтобы увидеть, как работает класс `ProjectCostVisitor`, необходимо воспользоваться классом `RunPattern`, исходный код которого представлен в листинге А. 128. Этот класс извлекает из файла хранящийся в нем проект, а затем перебирает все элементы этого проекта и вычисляет оценочную его стоимость.

Листинг А. 128. RunPattern.java

```
1. import java.io.File;
2. import java.util.ArrayList;
3. import java.util.Iterator;
4. public class RunPattern{
5.   public static void main(String [] arguments){
6.     System.out.println("Example for the Visitor pattern");
7.      System.out.println();
8.     System.out.println("This sample will use a ProjectCostVisitor to
   calculate");
9.     System.out.println(" the total amount required to complete a Project.");
10.     System.out.println();
11.
12.    System.out.println("Deserializing a test Project for Visitor pattern");
13.    System.out.println();
14.   if ( ! (new File("data.ser") .exists())){
15 .      DataCreator.serialize("data.ser");
16.    }
17.    Project project = (Project)(DataRetriever.deserializeData("data.ser"));
```

```
18.
19.    System.out.println("Creating a ProjectCostVisitor, to calculate the
   total cost of the project.");
20.    ProjectCostVisitor visitor = new ProjectCostVisitor ();
21.    visitor.setHourlyRate(100);
22.
23.    System.out.println("Moving through the Project, calculating total cost");
24.    System.out.println(" by passing the Visitor to each of the
   ProjectItems.");
25.    visitProjectItems(project, visitor);
26.    System.out.println("The total cost for the project is: " +
   visitor.getTotalCost() );
27.  }
28.
29.  private static void visitProjectItems(ProjectItem item, ProjectVisitor
   visitor){
30.    item.accept(visitor) ;
31.    if (item.getProjectItems() != null){
32.      Iterator subElements = item.getProjectItems().iterator ();
33.      while (subElements.hasNext ()){
34.         visitProjectItems((ProjectItem)subElements.next (), visitor);
35.      }
36.    }
37.  }
38.}
```

Template Method

В данном примере для иллюстрации того, как можно реализовать шаблон Template Method, используются классы PIM-приложения.

В качестве абстрактного класса, определяющего базовый метод, выбран класс ProjectItem (листинг А.129). Его метод getCostEstimate возвращает общую стоимость элемента проекта, которая вычисляется по следующей формуле:

стоимость = планируемое_время * почасовая_ставка + стоимость_материалов

Почасовая ставка определяется на уровне класса ProjectItem (для этого используются переменная rate, а также соответствующие методы getRate и setRate), но методы getTimeRequired и getMaterialsCost являются на уровне этого класса абстрактными. Это означает, что данные методы должны перекрываться подклассами, которые тем самым обеспечивают выполнение вычислений с учетом своей специфики.

Листинг А.129. ProjectItem.java

```
1. import java.io.Serializable;
2. public abstract class ProjectItem implements Serializable{
3.   private String name;
4.   private String description;
5.   private double rate;
6.
7.   public ProjectItem(){}
8.   public ProjectItem(String newName, String newDescription, double newRate){
9.      name = newName;
10.     description = newDescription;
11.     rate = newRate;
12.}
13.
```

```
14. public void setName(String newName){ name = newName; }
15. public void setDescription(String newDescription){ description =
    newDescription; }
16. public void setRate(double newRate){ rate = newRate; }
17.
18. public String getNameO { return name; }
19. public String getDescription (){ return description; }
20. public final double getCostEstimate(){
21.    return getTimeRequired() * getRate() + getMaterialsCost();
22. }
23. public double getRate(){ return rate; }
24.
25. public String toString(){ return getNameO; }
26.
27. public abstract double getTimeRequired();
28. public abstract double getMaterialsCost();
29.}
```

Класс Deliverable (листинг А. 130) представляет в приложении некоторый конкретный продукт, получаемый в ходе выполнения проекта. Так как продукт — это осязаемый физический предмет, методы getTimeRequired и getMaterialsCost класса возвращают заданные фиксированные значения.

Листинг А. 130. Deliverable.java

```
1. public class Deliverable extends ProjectItem{
2.  private double materialsCost;
3.  private double productionTime;
4.
5.  public Deliverable(){ }
6.  public Deliverable(String newName, String newDescription,
7.     double newMaterialsCost, double newProductionTime,
8.     double newRate){
9.    super(newName, newDescription, newRate);
10.   materialsCost = newMaterialsCost;
11.   productionTime = newProductionTime;
12. }
13.
14. public void setMaterialsCost(double newCost){ materialsCost = newCost; }
15. public void setProductionTime(double newTime){ productionTime = newTime; }
16.
17. public double getMaterialsCost(){ return materialsCost; }
18. public double getTimeRequired(){ return productionTime; }
19.}
```

Класс Task (листинг А. 131) представляет в приложении некую работу, которая может состоять из нескольких этапов или сводиться к получению нескольких продуктов. В этой связи метод getTimeRequired класса вычисляет общее время, требующееся для выполнения данной работы, путем перебора всех относящихся к ней элементов проекта с вызовом метода getTimeRequired каждого из этих элементов. То же самое выполняет и метод getMaterialsCost, с тем лишь отличием, что он вызывает метод getMaterialsCost каждого дочернего объекта.

Листинг А.131. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Task extends ProjectItem{
4.  private ArrayList projectItems = new ArrayList ();
5.  private double taskTimeRequired;
6.
7.  public Task() ( }
8.  public Task(String newName, String newDescription,
9.     double newTaskTimeRequired, double newRate){
10.    super (newName, newDescription, newRate);
11.    taskTimeRequired = newTaskTimeRequired;
12. }
13.
14. public void setTaskTimeRequired(double newTaskTimeRequired);
   taskTimeRequired = newTaskTimeRequired; }
15.  public void addProjectItem(ProjectItem element){
16.    if (!projectItems.contains(element)){
17.      projectItems.add(element);
18.    }
19. }
20. public void removeProjectItem(ProjectItem element){
21.    projectItems.remove(element);
22. }
23.
24. public double getTaskTimeRequired(){ return taskTimeRequired; }
25. public Iterator getProjectItemIterator(){ return projectItems.iterator(); }
26. public double getMaterialsCost (){
27.    double totalCost = 0;
28.    Iterator items = getProjectItemIterator();
29.    while (items.hasNext ()){
30.      totalCost += ((ProjectItem)items.next()).getMaterialsCost();
31.    }
32.    return totalCost;
33. )
34. public double getTimeRequired(){
35.    double totalTime = taskTimeRequired;
36.    Iterator items = getProjectItemIterator();
37.    while (items.hasNext()){
38.      totalTime += ((ProjectItem)items.next()).getTimeRequired();
39.    }
40.    return totalTime;
41. }
42.}
```

Класс RunPattern (листинг А.132) создает объект класса Deliverable и простую цепочку объектов класса Task, а затем вычисляет оценочную стоимость каждого этапа, вызывая метод getCostEstimate. Каждый объект использует заготовку, определенную классом Projectltem, но при вычислении требуемого времени и стоимости материалов применяет собственные методы.

Листинг А. 132. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.       System.out.println("Example for the Template Method pattern");
4.       System.out.println("This code demonstrates how the template method can");
5.       System.out.println(" be used to define a variable implementation for a");
6.       System.out.println(" common operation. In this case, the ProjectItem");
7.       System.out.println(" abstract class defines the method
   getCostEstimate,");
8.       System.out.println(" which is a combination of the cost for time and");
9.       System.out.println(" materials. The two concrete subclasses used here,");
10.      System.out.println(" Task and Deliverable, have different methods of");
11.      System.out.println(" providing a cost estimate.");
12.      System.out.println();
13.
14.      System.out.println("Creating a demo Task and Deliverable");
15.      System.out.println();
16.      Task primaryTask = new Task("Put a JVM on the moon", "Lunar mission as
   part of the JavaSpace program ; ) " , 240.0, 100.0);
17.      primaryTask.addProjectItem(new Task("Establish ground control", "",
   1000.0, 10.0));
18.      primaryTask.addProjectItem(new Task("Train the Javanaughts", "", 80.0, 30.0));
19.      Deliverable deliverableOne = new Deliverable("Lunar landing module",
   "Ask the local garage if they can make a few minor modifications to one of
   their cars", 2800, 40.0, 35.0);
20.
21.      System.out.println("Calculating the cost estimates using the Template
   Method, getCostEstimate.");
22.      System.out.println();
23.      System.out.println("Total cost estimate for: " + primaryTask);
24.      System.out.println("\t" + primaryTask.getCostEstimate());
25.      System.out.println();
26.
27.      System.out.println("Total cost estimate for: " + deliverableOne);
28.      System.out.println("\t" + deliverableOne.getCostEstimate());
29.  }
```

Структурныешаблоны

Adapter

В данном примере PIM-приложение использует программный интерфейс API, полученный из некоторого внешнего источника, который применяет в разработке программных компонентов язык, отличный от английского. Интерфейс API образуется двумя файлами, которые представляют собой набор готовых покупных классов, предназначенных для представления списка контактных лиц. Базовые операции определены в интерфейсе Chovnatlh (листинг А.133).

Листинг А.133. Chovnatlh.java

```
1. public interface Chovnatlh{
2.   public String tlhapWa$DIchPong();
3.   public String tlhapQavPong();
4.   public Strind tlhapPatlh();
5.   public String tlhapGhom();
6.
7.   public void cherWa$DIchPong(String chu$wa$DIchPong);
8.   public void cherQavPong(String chuSQavPong);
9.   public void cherPatlh(String chuSpatlh);
10.  public void cherGhom(String chu$ghom);
11. }
```

Реализация этих методов выполнена в тексте класса ChovnatlhImpl (листинг А. 134).

Листинг А.134. ChovnatlhImpl.java

```
1. //pong = name
2. //wa'DIch = first
3. //Qav = last
4. //patlh = rank (title)
5. //ghom = group (organization)
6. //tlhap = take (get)
7. //cher = set up (set)
8. //chu' = new
9. //chovnatlh = specimen (contact)
10.
11.public class ChovnatlhImpl implements Chovnatlh{
12. private String waSDIchPong;
13. private String QavPong;
14. private String patlh;
15. private String ghom;
16.
17. public ChovnatlhImpl(){ }
18. public ChovnatlhImpl(String chu$wa$DIchPong, String chu$QavPong,
19.    String chuSpatlh, String chu$ghom){
20.      waSDIchPong = chu$wa$DIchPong;
21.      QavPong = chuSQavPong;
22.      patlh = chuSpatlh;
23.      ghom = chu$ghom;
24. }
25.
26. public String tlhapWa$DIchPong(){ return waSDIchPong; }
27. public String tlhapQavPong(){ return QavPong; }
28. public String tlhapPatlh(){ return patlh; }
29. public String tlhapGhom(){ return ghom; }
```

```
30.
31. public void cherWa$DIchPong(String chu$wa$DIchPong){ wa$DIchPong =
   chu$wa$DIchPong; )
32. public void cherQavPong(String chu$QavPong){ QavPong = chu$QavPong; }
33. public void cherPatlh(String chu$patlh){ patlh = chu$patlh; }
34. public void cherGhom(String chu$ghom){ ghom = chu$ghom; }
35.
36. public String toString(){
37.    return wa$DIchPong + " " + QavPong + ": " +patlh + ", " + ghom;
38. }
39.}
```

Для того чтобы привести в соответствие методы класса `ChovnatlhImpl` с методами класса `Contact` (листинг А.135), следует воспользоваться объектом-адаптером. Эту задачу решает класс `ContactAdapter` (листинг А.136), внутренняя переменная которого содержит ссылку на объект класса `Chovnatlhlmpl`. Объект класса `ContactAdapter` управляет данными, образующими информацию о контактном лице: имя, должность и организация, которую это лицо представляет.

Листинг А.135. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitleO;
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А.136. ContactAdapter.java

```
1. public class ContactAdapter implements Contact(
2.   private Chovnatlh contact;
3.
4.   public ContactAdapter(){
5.      contact = new ChovnatlhImpl();
6.   }
7.   public ContactAdapter(chovnatlh newContact){
8.      contact = newContact;
9.   }
10.
11.  public String getFirstName(){
12.     return contact.tlhapWa$DIchPong() ;
13.  }
14.  public String getLastName(){
15.     return contact.tlhapQavPong();
16.  }
17.  public String getTitle(){
18.     return contact.tlhapPatlh();
19.  }
```

```
20. public String getOrganization(){
21.    return contact.tlhapGhom() ;
22. }
23.
24. public void setContact(Chovnatlh newContact){
25.    contact = newContact;
26. }
27. public void setFirstName(String newFirstName){
28.    contact.cherWa$DIchPong(newFirstName);
29. }
30. public void setLastName(String newLastName(String newLastName)}
31.    contact.cherQavPong(newLastName);
32. }
33. public void setTitle(String newTitle){
34.    contact.cherPatlh(newTitle);
35. }
36. public void setOrganization(String newOrganization))
37.    contact.cherGhom(newOrganization) ;
38. }
39.
40. public String toString (){
41.    return contact.toString();
42. )
43.}
```

Использование на практике объекта-адаптера можно увидеть на примере класса RunPattern (листинг А. 137). Вначале класс RunPattern создает экземпляр класса ContactAdapter, а затем с его помощью создает тестовый экземпляр класса Contact. Однако при вызове метода toString, объявленного в классе Contact-Adapter, текстовое представление контактной информации возвращается классу RunPattern классом ChovnatlhImpl, который на самом деле обеспечивает хранение этой информации.

Листинг А.137. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.      System.out.println("Example for the Adapter pattern");
4.      System.out.println();
5.      System.out.println("This example will demonstrate the Adapter by using
   the");
6.      System.out.println(" class ContactAdapter to translate from classes
   written");
7.      System.out.println (" in a foreign language (Chovnatlh and
   Chovnatlhlmpl),");
8.      System.out.println(" enabling their code to satisfy the Contact
   interface.");
9.      System.out.println();
10.
11.     System.out.println ("Creating a new ContactAdapter. This will, by extension,");
12.     System.out.println (" create an instance of Chovnatlhlmpl which will
   provide");
13.     System.out.println(" the underlying Contact implementation.");
14.     Contact contact = new ContactAdapter();
15.     System.out.println();
16.
17.     System.out.println("ContactAdapter created. Setting contact data.");
18.     contact.setFirstName("Thomas");
19.     contact.setLastName("Williamson");
```

```
20.    contact.setTitle("Science Officer");
21.    contact.setOrganization("W3C");
22.    System.out.println();
23.
24.    System.out.println("ContactAdapter data has been set. Printing out
   Contact data.");
25.    System.out.println();
26.   System.out.println(contact.toString() ) ;
27. }
28.}
```

Bridge

В данном примере показано, как использовать шаблон Bridge для расширения функциональности списка неотложных дел PIM-приложения. Сам по себе этот список очень прост и представляет собой обычный объект с возможностью добавления и удаления элементов типа String.

В соответствии с шаблоном Bridge элемент разделяется на две части: абстракцию и реализацию. Реализация— это класс, который выполняет всю работу (в данном примере, он сохраняет элементы в списке и обеспечивает их получение). Общее поведение списка PIM-приложения определяется интерфейсом ListImpl (листинг А.138).

Листинг А.138. ListImpl.java

```
1. public interface ListImpl{
2.  public void addItem(String item);
3.  public void addItem(String item, int position);
4.  public void removeItem(String item);
5.  public int getNumberOfItems ();
6.  public String getItem(int index);
7.  public boolean supportsOrdering();
8. }
```

Класс OrderedListImpl, представленный в листинге А.139, реализует интерфейс ListImpl и сохраняет элементы списка во внутреннем объекте ArrayList.

Листинг А.139. OrderedListImpl.java

```
1. import java.util.ArrayList;
2. public class OrderedListImpl implements ListImpl{
3.  private ArrayList items = new ArrayList();
4.
5.  public void addItem(String item)(
6.     if (!items.contains(item){
7.       items.add(item);
8.     }
9.  )
10. public void addItem(String item, int position){
11.    if (!items.contains(item)){
12.      items.add(position, item);
13.    }
14. }
15.
```

```
16. public void removeItem(String item){
17.    if (items.contains(item)){
18.      items.remove(items.indexOf(item));
19.    }
20. }
21.
22. public boolean supportsOrdering(){
23.    return true;
24. }
25.
26. public int getNumberOfItems(){
27.   return items.size();
28. }
29.
30. public String getItem(int index){
31.    if (index < items.size()){
32.      return (String)items.get(index);
33.    }
34.    return null;
35. }
36.}
```

Абстракция представляет собой перечень выполняемых над списком операций, который доступен внешнему миру. Класс BaseList (листинг А.140) содержит объявление основных возможностей списка.

Листинг А.140. BaseList.java

```
1. public class BaseList{
2.   protected ListImpl implementor;
3.
4.   public void setImplementor(ListImpl impl){
5.      implementor = impl;
6.   }
7.
8.   public void add(String item){
9.      implementor.addItem(item);
10.  }
11.  public void add(String item, int position){
12.     if (implementor.supportsOrdering(){
13.       implementor.addItem(item, position);
14.     }
15.  )
16.
17.  public void remove(String item){
18.     implementor.removeItem(item);
19.  }
20.
21.  public String get(int index){
22.     return implementor.getItem(index);
23.  }
24.
25.  public int count(){
26.     return implementor.getNuberOfItems();
27.  }
28.}
```

Обратите внимание на то, что все операции делегированы переменной implementor, представляющей реализацию списка. Какая бы операция не запрашивалась, она в действительности делегируется базовым классом связанному с ним классу конкретной реализации списка.

Расширить возможности базового класса очень просто: достаточно создать подкласс и добавить в него нужную функциональность. Приведенный в листинге А.141 класс NumberedList наглядно демонстрирует всю мощь шаблона **Bridge** – достаточно просто перекрыть метод get, и класс уже может обеспечить работу с нумерованными списками.

Листинг А.141. NumberedList.java

```
1. public class NumberedList extends BaseList{
2.   public String get(int index){
3.     return (index + 1) + ". " + super.get(index) ;
4.   }
```

Еще одна абстракция представлена классом OrnamentedList (листинг А.142). В данном случае расширение функциональности заключается в том, что каждый элемент списка предваряется заданным символом.

Листинг А.142. OrnamentedList.java

```
1. public class OrnamentedList extends BaseList(
2.   private char itemType;
3.
4.   public char getItemType(){ return itemType; }
5.   public void setItemType(char newItemType){
6.     if (newItemType > ' '){
7.       itemType = newItemType;
8.     )
9.   }
10.
11.  public String get(int index){
12.     return itemType + " " + super.get(index);
13.   }
14.}
```

Запуск данного примера на выполнение осуществляет класс RunPattern (листинг А.143). Метод main класса создает объект класса OrderedListImpl и наполняет полученный список элементами. Затем он связывает реализацию класса с тремя различными объектами абстракций и выводит на экран содержимое списка. Здесь можно наблюдать реализацию двух важных принципов: во-первых, с одной реализацией класса можно применять несколько абстракций, а во-вторых, каждая абстракция может по-своему модифицировать представление нижележащих данных.

Листинг А. 143. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments){
3.     System.out.println("Example for the Bridge pattern");
4.      System.out.println();
5.     System.out.println("This example divides complex behavior among two");
6.     System.out.println(" classes - the abstraction and the implementation.");
7.     System.out.println ();
8.     System.out.println("In this case, there are two classes which can
   provide the");
9.     System.out.println(" abstraction - BaseList and OrnamentedList.
   The BaseList");
10.    System.out.println(" provides core functionality, while the
   OrnamentedList");
11.    System.out.println(" expands on the model by adding a list character.");
12.     System.out.println();
13.    System.out.println("The OrderedListImpl class provides the underlying
   storage");
14.    System.out.println(" capability for the list, and can be flexibly paired
   with");
15.    System.out.println(" either of the classes which provide the abstraction.");
16.
17.    System.out.println("Creating the OrderedListImpl object.");
18.    ListImpl implementation = new OrderedListImpl ();
19.
20.    System.out.println("Creating the BaseList object.");
21.    BaseList listOne = new BaseList();
22.    listOne.setImplementor(implementation) ;
23.     System.out.println();
24.
25.    System.out.println("Adding elements to the list.");
26.     listOne.add("One");
27.     listOne.add("Two");
28.    listOne.add("Three") ;
29.     listOne.add("Four");
30.     System.out.println();
31.
32.    System.out.println("Creating an OrnamentedList object.");
33.    OrnamentedList listTwo = new OrnamentedList();
34.    listTwo.setImplementor(implementation) ;
35.     listTwo.setItemType('+');
36.     System.out.println();
37.
38.    System.out.println("Creating an NumberedList object.");
39.    NumberedList listThree = new NumberedList();
40.    listThree.setImplementor(implementation) ;
41.     System.out.println();
42.
43.    System.out.println("Printing out first list (BaseList)");
44.    for (int i = 0; i < listOne.count(); i++){
45.      System.out.println("\t" + listOne.get(i));
46.     }
47.    System.out.println() ;
48.
49.    System.out.println("Printing out second list (OrnamentedList)");
50.     for (int i = 0; i < listTwo.count(); i++) {
51.       System.out.println("\t" + listTwo.get(i));
52.     }
53.     System.out.println();
54.
55.     System.out.println("Printing our third list (NumberedList)");
```

```
56.    for (int i = 0; i < listThree.count(); i++){
57.      System.out.println("\t" + listThree.get(i));
58.    }
59.  }
60.)
```

Composite

В данном примере показано, как с помощью шаблона Composite можно обеспечить определение времени, необходимого для выполнения всего проекта или его отдельного этапа. Пример состоит из четырех основных составляющих.

- Deliverable. Класс, представляющий конечный продукт, получаемый в результате завершения текущего этапа.
- `Project`. Класс, являющийся корнем композиции и представляющий весь проект в целом.
- `ProjectItem`. Интерфейс, описывающий общую для всех элементов проекта функциональность. Именно в этом интерфейсе содержится объявление метода `getTimeRequired`.
- Task. Класс, представляющий коллекцию операций, подлежащих выполнению. Каждый экземпляр этого класса содержит ссылку на коллекцию объектов класса Projectltem.

Общая функциональность, которой должен обладать любой объект проекта, определяется в интерфейсе `ProjectItem` (листинг А.144). В рассматриваемом примере в этом интерфейсе содержится объявление лишь одного метода getTimeRequired.

Листинг А. 144. Projectltem. java

```
1. import java.io.Serializable;
2. public interface Projectltem extends Serializable{
3.   public double getTimeRequired();
4. }
```

Поскольку элементы проекта могут образовывать древовидную структуру, классы Deliverable и Task реализуют интерфейс Projectltem. Класс Deliverable (листинг А.145) представляет собой терминальный узел, который не может ссылаться на другие элементы проекта.

Листинг А.145. Deliverable.java

```
1. public class Deliverable implements ProjectItem{
2.   private String name;
3.   private String description;
4.   private Contact owner;
5.
6.   public Deliverable(){ }
7.   public Deliverable(String newName, String newDescription,
8.      Contact newOwner){
```

```
9.     name = newName;
10.    description = newDescription;
11.    owner = newOwner;
12. }
13.
14. public String getName(){ return name; }
15. public String getDescription(){ return description; }
16. public Contact getOwner(){ return owner; }
17. public double getTimeRequired(){ return 0; }
18.
19. public void setName(String newName){ name = newName; }
20. public void setDescription(String newDescription){ description =
    newDescription; }
21. public void setOwner(Contact newOwner){ owner = newOwner; }
22.}
```

Классы Project (листинг А. 146) и Task (листинг А.147), напротив, являются нетерминальными узлами и поэтому оба сохраняют ссылку на коллекцию элементов проекта, представляющих дочерние этапы проекта Task и связанные с ними продукты Deliverable.

Листинг А.146. Project.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Project implements ProjectItem{
4.   private String name;
5.   private String description;
6.   private ArrayList projectItems = new ArrayList();
7.
8.  public Project (){ }
9.  public Project(String newName, String newDescription){
10.    name = newName;
11.    description = newDescription;
12.  }
13.
14.  public String getName(){ return name; }
15.  public String getDescription(){ return description; }
16.  public ArrayList getProjectItems(){ return projectItems; }
17.  public double getTimeRequired(){
18.    double totalTime = 0;
19.    Iterator items = projectItems.iterator();
20.    while(items.hasNext(){
21.      ProjectItem item = (ProjectItem)items.next();
22.      totalTime += item.getTimeRequired();
23.    }
24.    return totalTime;
25.  }
26.
27.  public void setName(String newName){ name = newName; }
28.  public void setDescription(String newDescription)) description =
    newDescription; }
29.
30.  public void addProjectItem(ProjectItem element))
31.    if (!projectItems.contains(element)){
32.      projectItems.add(element) ;
33.    )
34.  }
```

```
35.  public void removeProjectItem(ProjectItem element){
36.     projectItems.remove(element);
37.  }
38.}
```

Листинг A.147. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator:
3. public class Task implements ProjectItem{
4.   private String name;
5.   private String details;
6.   private ArrayList projectltems = new ArrayList ();
7.   private Contact owner;
8.   private double timeRequired;
9.
10. public Task() { }
11.  public Task(String newName, String newDetails,
12.     Contact newOwner, newTimeRequired)(
13.     name = newName;
14.     details = newDetails;
15.     owner = newOwner;
16.     timeRequired = newTimeRequired;
17.  }
18.
19.  public String getName(){ return name; }
20.  public String getDetails ()( return details; )
21.  public ArrayList getProjectItems(){ return projectltems; }
22.  public Contact getOwner(){ return owner; }
23.  public double getTimeRequired(){
24.     double totalTime = timeRequired;
25.     Iterator items = projectItems.iterator();
26.    while (items.hasNext()){
27.       ProjectItem item = (Projectltem) items .next();
28.       totalTime += item. getTimeRequired ();
29.     }
30.     return totalTime;
31.  }
32.
33.  public void setName(String newName)[ name = newName; }
34.  public void setDetails(String newDetails){ details = newDetails; }
35.  public setOwner(Contact newOwner)( owner = newOwner; }
36.  public void setTimeRequired(double newTimeRequired){ timeRequired =
   newTimeRequired; }
37.
38.  public void  addProjectItem(ProjectItem element){
39.     if (!projectItems.contains(element)){
40.       projectItems.add(element);
41.     }
42.  }
43.  public void removeProjectItem(ProjectItem element){
44.    projectItems.remove(element) ;
45.  }
46.}
```

Лучше всего исследовать работу шаблона Composite в процессе изучения работы метода `getTimeRequired`. Для того чтобы получить оценку временных затрат для любой части проекта, достаточно просто вызвать метод getTimeRequired объекта Task (если нужна оценка для всего проекта, следует вызвать аналогичный метод объекта Project). В зависимости от реализации этот метод возвращает одно из следующих значений.

- Deliverable. Возвращает 0.
- Project или Task. Возвращает значение, представляющее временные затраты, необходимые для выполнения текущего этапа, плюс значение, представляющее собой сумму всех значений, полученных при вызове метода getTimeRequired всех элементов проекта, связанных с текущим элементом.

Вспомогательный программный код данного примера состоит из интерфейса Contact (листинг А. 148) и класса `ContactImpl` (листинг А. 149), которые обеспечивают представление владельца этапа проекта или продукта, полученного на определенном этапе.

Листинг А. 148. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization) ;
13.}
```

Листинг А. 149. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14.  }
15.
16.  public String getFirstName(){ return firstName; }
17.  public String getLastName(){ return lastName; }
18.  public String getTitle(){ return title; }
```

```
19. public String getOrganization ()( return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization;}
25.
26. public String toString(){
27.    return firstName + SPACE + lastName;
28. }
29.}
```

В данном примере используется небольшой демонстрационный проект, иллюстрирующий применение шаблона Composite на практике. Для упрощения задачи управления хранящейся в файле информацией проекта класс DataCreator (листинг A.150) создает пример проекта и сериализует его в файл.

Листинг A. 150. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. public class DataCreator{
6.  private static final String DEFAULT_FILE = "data.ser";
7.
8.  public static void main(String [] args){
9.     String fileName;
10.    if (args.length == 1) {
11.      fileName = args[0];
12.    }
13.    else{
14.      fileName = DEFAULT_FILE;
15.    }
16.    serialize(fileName);
17. }
18.
19. public static void serialize(String fileName){
20.    try{
21.      serializeToFile(createData(), fileName);
22.    }
23.    catch (IOException exc) {
24.      exc.printStackTrace() ;
25.    }
26. }
27.
28. private static Serializable createData(){
29.    Contact contactl = new ContactImpl("Dennis", "Moore", "Managing
    Director", "Highway Man, LTD");
30.    Contact contact2 = new ContactImpl("Joseph", "Mongolfier", "High Flyer",
    "Lighter than Air Productions");
31.    Contact contact3 = new ContactImpl("Erik", "Njoll", "Nomad without
    Portfolio", "Nordic Trek, Inc.");
32.    Contact contact4 = new ContactImpl("Lemming", "", "Principal
    Investigator", "BDA");
33.
34.    Project project = new Project("IslandParadise", "Acquire a personal
    islandparadise");
35.    Deliverable deliverable1 = new Deliverable("Island Paradise", "", contactl);
```

```
36.  Task task1 = new Task("Fortune", "Acquire a small fortune", contact4, 0);
37.  Task task2 = new Task("Isle", "Locate an island for sale", contact2, 7.5);
38.  Task task3 = new Task("Name", "Decide on a name for the island", contact3, 2);
39.   project.addProjectItem(deliverable1);
40.   project.addProjectItem(task1);
41.   project.addProjectItem(task2);
42.   project.addProjectItem(task3);
43.
44.   Deliverable deliverable11 = new Deliverable("$1,000,000", "(total net
   worth after taxes)", contact1);
45.   Task task11 = new Task("Fortune1", "Use psychic hotline to predict
   winning lottery numbers", contact4, 2.5);
46.   Task task12 = new Task("Fortune2", "Invest winnings to ensure 50% annual
   interest", contact1, 14.0);
47.   task1.addProjectItem(task11) ;
48.   task1.addProjectItem(task12) ;
49.    task1.addProjectItem(deliverable11);
50.
51.   Task task21 = new Task("Isle1", "Research whether climate is better in
   the Atlantic or Pacific", contact1, 1.8);
52.   Task task22 = new Task("Isle2", "Locate an island for auction on EBay",
   contact4, 5.0);
53.   Task task23 = new Task("Isle2a", "Negotiate for sale of the island",
   contact3, 17.5);
54.    task2.addProjectItem(task21);
55.    task2.addProjectItem(task22);
56.    task2.addProjectItem(task23);
57.
58.  Deliverable deliverable31 = new Deliverable("Island Name", "", contact1);
59.    task3.addProjectItem(deliverable31);
60.    return project;
61. }
62.
63. private static void serializeToFile(Serializable content, String fileName)
   throws IOException{
64.   ObjectOutputStream serOut = new ObjectOutputStream(new
   FileOutputStream(fileName));
65.   serOut.writeObject(content) ;
66.    serOut.close();
67. }
68.}
```

Класс `DataRetriever` обеспечивает десериализацию объекта из файла, выполняемую с помощью метода `deserializeData` (листинг А. 151).

Листинг А.151. DataRetriever.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5.
6. public class DataRetriever{
7.  public static Object deserializeData(String fileName){
8.    Object returnValue = null;
9.     try{
10.      File inputFile = new File(fileName);
11.      if (inputFile.exists() && inputFile.isFile()){
12.          ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(fileName));
13.        returnValue = readIn.readObject ();
```

```
14.         readIn.close () ;
15.       }else{
16.         System.err.println("Unable to locate the file " + fileName);
17.       }
18.     }catch (ClassNotFoundException exc){
19.       exc.printStackTrace();
20.
21.     )catch (IOException exc){
22.       exc.printStackTrace();
23.     }
24.     return returnValue;
25. }
26.}
```

Класс RunPattern (листинг А.152) сначала использует экземпляр класса DataRetriever для десериализации проекта, а затем вызывает метод getTimeRequired для определения времени, необходимого для выполнения всего проекта.

Листинг А.152. RunPattern.java

```
1. import java.io.File;
2. public class RunPattern{
3.   public static void main(String [] arguments){
4.     System.out.println("Example for the Composite pattern");
5.      System.out.println();
6.     System.out.println("This code sample will propagate a method call
   throughout");
7.     System.out.println(" a tree structure. The tree represents a project,
   and is");
8.     System.out.println(" composed of three kinds of ProjectItems -
   Project, Task,");
9.     System.out.println(" and Deliverable. Of these three classes,
   Project and Task");
10.    System.out.println(" can store an ArrayList of ProjectItems.
   This means that");
11.    System.out.println(" they can act as branch nodes for our tree.
   The Deliverable");
12.    System.out.println(" is a terminal node, since it cannot hold any
   ProjectItems.");
13.     System.out.println();
14.    System.out.println("In this example, the method defined by
   ProjectItem,");
15.    System.out.println(" getTimeRequired, provides the method to demonstrate
   the");
16.    System.out.println(" pattern. For branch nodes, the method will be
   passed on");
17.   System.out.println(" to the children. For terminal nodes (Deliverables), a");
18.    System.out.println(" single value will be returned.");
19.     System.out.println();
20.    System.out.println("Note that it is possible to make this method call
   ANYWHERE");
21.    System.out.println(" in the tree, since all classes implement the
   getTimeRequired");
22.    System.out.println(" method. This means that you are able to calculate
   the time");
23.    System.out.println(" required to complete the whole project OR any part
   of it.");
24.     System.out.println();
25.
26.    System.out.println("Deserializing a test Project for the Composite pattern");
```

```
27.   System.out.println ();
28.    if (!(new File("data.ser").exists())) {
29.      DataCreator.serialize("data.ser");
30.    }
31.    Project project = (Project)(DataRetriever.deserializeData("data.ser"));
32.
33.    System.out.println("Calculating total time estimate for the project");
34.    System.out.println("\t" + project.getDescription());
35.    System.out.println("Time Required: " + project.getTimeRequired());
36.
37. }
38.}
```

Decorator

В данном примере показано, как с помощью шаблона Decorator можно расширить возможности объектов, представляющих элементы проекта. Основой программного представления проекта является интерфейс Projectltem (листинг А.153). Этот интерфейс должен реализовываться любым классом, который будет использован в проекте. В данном примере интерфейс `ProjectItem` содержит определение лишь одного метода `getTimeRequired`.

Листинг А.153. Projectltem.**java**

```
1. import java.io.Serializable;
2. public interface Projectltem extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public double getTimeRequired();
5. }
```

Интерфейс Projectltem реализуется классами Deliverable (листинг А.154) и Task (листинг А.155), в которых содержится определение базовой функциональности двух типов элементов проекта. Как и в предыдущих примерах, класс Task представляет определенный этап проекта, а класс Deliverable— какой-то конкретный продукт.

Листинг А.154. Deliverable.java

```
1. public class Deliverable implements ProjectItem{
2.   private String name;
3.   private String description;
4.   private Contact owner;
5.
6.   public Deliverable (){ }
7.   public Deliverable(String newName, String newDescription,
8.      Contact newOwner){
9.      name = newName;
10.     description = newDescription;
11.     owner = newOwner;
12.  }
13.
```

```
14. public String getName(){ return name; }
15. public String getDescription(){ return description; }
16. public Contact getOwner(){ return owner; }
17. public double getTimeRequired(){ return 0; }
18.
19. public void setName(string newName){ name = newName; }
20. public void setDescription(String newDescription){ description =
    newDescription; }
21. public void setOwner(Contact newOwner){ owner = newOwner; }
22.
23. public String toString(){
24.   return "Deliverable: " + name;
25. }
26. }
```

Листинг А. 155. Task.java

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class Task implements ProjectItem{
4.   private String name;
5.   private ArrayList projectItems = new ArrayList ();
6.   private Contract owner;
7.   private double timeRequired;
8.
9.   public Task() { }
10.  public Task(String newName, Contact newOwner,
11.    double newTimeRequired) {
12.    name = newName;
13.    owner = newOwner;
14.    timeRequired = newTimeRequired;
15. }
16.
17. public String getName(){ return name; }
18. public ArrayList getProjectItems(){ return projectItems; }
19. public Contact getOwner(){ return owner; }
20. public double getTimeRequired(){
21.    double totalTime = timeRequired;
22.    Iterator items = projectItems.iterator();
23.    while(items.hasNext()){
24.      ProjectItem = (ProjectItem)items.next();
25.      totalTime += item.getTimeRequired();
26.    }
27.    return totalTime;
28. }
29.
30. public void setName(String newName){ name = newName; }
31. public void setOwner(Contact newOwner){ owner = newOwner; }
32. public void setTimeRequired(double newTimeRequired){ timeRequired =
    newTimeRequired; }
33.
34. public void addProjectItem(ProjectItem element){
35.    if (!projectItems.contains(element)){
36.      projectItems.add(element);
37.    }
38. }
39. public void removeProjectItem(ProjectItem element){
40.    projectItems.remove(element);
41. }
```

```
42.
43. public String toString(){
44.    return "Task: " + name;
45. }
46. }
```

Теперь настало время расширить базовые средства этих классов. Класс ProjectDecorator (листинг А. 156) обеспечивает принципиальную возможность дополнения классов Task и Deliverable.

Листинг А. 156. ProjectDecorator.java

```
1. public abstract class ProjectDecorator implements ProjectItem{
2.   private ProjectItem projectItem;
3.
4.   protected ProjectItem getProjectItem(){ return projectItem; }
5.   public void setProjectItem(ProjectItem newProjectItem){ projectItem =
   newProjectItem; }
6.
7.   public double getTimeRequired(){
8.      return projectItem.getTimeRequired();
9.   }
10.}
```

Класс ProjectDecorator реализует интерфейс ProjectItem и содержит переменную на следующий экземпляр класса ProjectItem, представляющий "декорируемый" элемент. Необходимо отметить, что класс ProjectDecorator делегирует вызов метода getTimeRequired своему внутреннему элементу. Это можно проделать и с любым другим методом, который зависит от функциональности нижележащего компонента. Действительно, если на выполнение этапа проекта отводится пять дней, метод должен вернуть значение, соответствующие пяти дням, вне зависимости от того, какие дополнительные возможности были добавлены к классу Task путем "декорирования".

В данном примере приведено два подкласса класса ProjectDecorator, демонстрирующих, как можно добавить дополнительную функциональность к элементам проекта. Класс DependentProjectItem (листинг А. 157) использован для того, чтобы показать, что объект класса Task или Deliverable зависит от завершения какого-то другого элемента проекта, представленного объектом класса ProjectItem.

Листинг А.157. DependentProjectItem.java

```
1. public class DependentProjectItem extends ProjectDecorator{
2.   private ProjectItem dependentItem;
3.
4.   public DependentProjectItem(){ }
5.   public DependentProjectItem(ProjectItem newDependentItem)[
6.      dependentItem = newDepentItem;
7.   }
8.
9.   public ProjectItem getDependentItem(){ return dependentItem; }
10.
11.  public void setDependentItem(ProjectItem newDependentItem){ dependentItem =
   newDependentItem;  }
```

```
12.
13. public String toString(){
14.    return getProjectItem() .toString() + EOL_STRING
15.     + "\tProjectItem dependent on: " + dependentItem;
16. }
17.}
```

Класс SupportedProjectItem (листинг А.158) "декорирует" класс ProjectItem и сохраняет ссылку на коллекцию ArrayList вспомогательных документов— файловых объектов, представляющих дополнительную информацию или ресурсы.

Листинг А.158. SupportedProjectItem.java

```
1. import java.util.ArrayList;
2. import java.io.File;
3. public class SupportedProjectltem extends ProjectDecorator{
4.  private ArrayList supportingDocuments = new ArrayList ();
5.
6.  public SupportedProjectItem(){ }
7.  public SupportedProjectItem(File newSupportingDocument){
8.     addSupportingDocument(newSupportingDocument);
9.  }
10.
11. public ArrayList getSupportingDocuments(){
12.    return supportingDocuments(){
13. }
14.
15. public void addSupportingDocument(File document){
16.    if (!supportingDocuments.contains(document)){
17.      supportingDocuments.add(document);
18.    )
19. }
20.
21. public void removeSupportingDocument(File document){
22.    supportingDocuments.remove(document);
23. )
24.
25. public String toString(){
26.   return getProjectltemO .toString() + EOL_STRING
27.     + "\tSupporting Documents: " + supportingDocuments;
28. )
29.}
```

Достоинством именно такого способа определения дополнительных возможностей является то, что он облегчает создание элементов проекта, обладающих некоторым набором этих возможностей. Используя эти классы, можно создать этап проекта, зависящий от другого элемента проекта, или этап со вспомогательными документами. Можно также объединить все "декорирующие" классы в цепочку и создать этап, который зависит от другого этапа и имеет вспомогательные документы. Эта гибкость и является самой сильной характеристикой шаблона Decorator.

Вспомогательный программный код данного примера состоит из интерфейса Contact (листинг А.159) и класса ContactImpl (листинг А.160), которые обеспечивают представление владельца этапа проекта или продукта, полученного на определенном этапе.

Листинг А. 159. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitleO;'
7.   public String getOrganization ();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг А. 160. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName)) firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; )
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization;}
25.
26. public String toStringOf
27.    return firstName + SPACE + lastName;
28. }
29.)
```

Класс `RunPattern` (листинг А. 161) создает несколько экземпляров класса `ProjectItern` и выводит на экран их строковое представление. Затем он создает несколько экземпляров класса `ProjectDecorator`, связывает их с объектами `Task`, вызывая методы `setProjectItem` каждого экземпляра, и отображает строковое представление обновленных объектов класса `Task`.

Листинг A. 161. RunPattern.java

```
1. import java.io.File;
2. public class RunPattern{
3.   public static void main(String [] arguments)(
4.     System.out.println("Example for the Decorator pattern");
5.    System. out .println ();
6.     System.out.println("This demonstration will show how Decorator classes
   can be used");
7.     System.out.println(" to extend the basic functionality of ProjectItems.
   The Task and");
8.     System.out.println(" Deliverable classes provide the basic Projectltems,
   and their");
9.     System.out.println(" functionality will be extended by adding subclasses
   of the");
10.    System.out.println(" abstract class ProjectDecorator.");
11.   System, out .println ();
12.    System.out.println("Note that the toString method has been overridden
   for all Projectltems,");
13.    System.out.println(" to more effectively show how Decorators are
   associated with their");
14.    System.out.println(" Projectltems.");
15.    System.out.println();
16.
17.    System.out.println("Creating Projectltems.");
18.   Contact contactl = new ContactImpl ("Simone", "Roberto", "Head Researcher
   and Chief Archivist", "Institute for Advanced (Java) Studies");
19.    Task task1 = new Task("Perform months of diligent research", contactl,
   20.0);
20.    Task task2 = new Task("Obtain grant from World Java Foundation",
   contactl, 40.0);
21.    Deliverable deliverablel = new Deliverable("Java History",
   "Comprehensive history of the design of all Java APIs", contactl);
22.    System.out.println("ProjectItem objects created. Results:");
23.    System.out.println(task1);
24.    System.out.println(task2);
25.    System.out.println(deliverable1);
26.    System.out.println();
27.
28.    System.out.println("Creating decorators");
29.    ProjectDecorator decorator1 = new SupportedProjectItem(new
   File("JavaHistory.txt"));
30.    ProjectDecorator decorator2 = new DependentProjectItem (task2) ;
31.    System.out.println("Decorators created. Adding decorators to the first
   task");
32.    decorator1.setProjectItem(task1);
33.    decorator2.setProjectItem(decorator1);
34.    System.out.println() ;
35.    System.out.println("Decorators added. Results");
36.    System.out.println(decorator2) ;
37.
38.    System.out.println("");
39. }
40.}
```

Facade

Для того чтобы PIM-приложение было с точки зрения пользователей более функциональным, нужно дать им возможность настраивать его параметры. К таким настраиваемым параметрам можно отнести, например, гарнитуру и размер шрифта, цветовую гамму, порядок запуска отдельных подсистем, используемую по умолчанию денежную единицу и т.п. В данном примере рассмотрен набор параметров, определяющих представление в системе национальных стандартов даты, времени, валюты и т.п.

Роль класса Facade в рассматриваемом примере играет класс InternationalizationWizard (листинг А.162). Этот класс координирует взаимодействие клиента с несколькими объектами, так или иначе связанными с выбранным национальным стандартом.

Листинг А.162. InternationalizationWizard.java

```
1. import java.util.HashMap;
2. import java.text.NumberFormat;
3. import java.util.Locale;
4. public class InternationalizationWizard{
5.   private HashMap map;
6.   private Currency currency = new CurrencyO;
7.   private InternationalizedText propertyFile = new InternationalizedText();
8.
9.   public InternationalizationWizard() {
10.     map = new HashMap();
11.    Nation[] nations = {
12.      new Nation("US", 'S', "+1", "us .properties",
   NumberFopmat.getInstance(Locale.US)),
13.      new Nation("The Netherlands", 'f', "+31", "dutch.properties",
   NumberFormat.getInstance(Locale.GERMANY)),
14.      new Nation("France", 'f', "+33", "french.properties",
   NumberFormat.getInstance(Locale.FRANCE))
15.     };
16.    for (int i = 0; i < nations.length; i++) {
17.      map.put(nations[i].getName(), nations[i]);
18.     }
19.  }
20.
21.  public void setNation(string name) {
22.     Nation nation = (Nation)map.get(name);
23.     if (nation != null) {
24.      currency.setCurrencySymbol(nation.getSymbol() ) ;
25.      currency.setNumberFormat(nation.getSymbol() ) ;
26.      PhoneNumber.setSelectedInterPrefix(nation.getDialingPrefix() ) ;
27.       propertyFile.setFileName(nation.getPropertyFileName());
28.     }
29.  }
30.
31.  public Object[] getNations(){
32.    return map. values (). toArray ();
33.  }
34.  public Nation getNation(String name)[
35.     return (Nation)map.get(name);
36.  }
37.  public char getCurrencySymbol(){
```

```
38.    return currency.getCurrencySymbol();
39. }
40. public NumberFormat getNumberFormat(){
41.    return currency.getNumberFormat(){
42. }
43. public String getPhonePrefix(){
44.    return ProneNumber.getSelectedInterPrefix();
45. }
46. public String getProperty(string key){
47.    return propertyFile.getProperty(key);
48. }
49. public String getProperty(String key, String defaultValue){
50.    return propertyFile.getProperty(key, defaultValue);
51. }
52. }
```

Обратите внимание на то, что класс `InternationalizationWizard` имеет несколько методов вида getXXXX, с помощью которых он осуществляет делегирование вызовов ассоциированным объектам. Кроме того, класс имеет метод setNation, используемый для изменения типа национального стандарта в соответствии с предпочтениями пользователя.

Хотя управление параметрами представления национального стандарта, как показано в данном примере, осуществляется через фасадный класс `InternationalizationWizard`, при этом сохраняется возможность управления каждым объектом в отдельности. Эта особенность является одним из преимуществ данного шаблона— он позволяет в некоторых случаях управлять группой объектов, но вместе с тем не ограничивает свободы по раздельному управлению компонентами.

Вызов метода setNation фасадного класса InternationalizationWizard используется для установки типа выбранного национального стандарта. Выполнение этой операции ведет к изменению параметров классов Currency (листинг A.163), `PhoneNumber` (листинг A. 164) и `InternationalizedText` (листинг A.165).

Листинг A. 163. `Currency.java`

```
1. import java.text.NumberFormat;
2. public class Currency{
3.  private char currencySymbol;
4.  private NumberFormat numberFormat;
5.
6.  public void setCurrencySymbol(char newCurrencySymbol){ currencySymbol =
   newCurrencySymbol; }
7.  public void setNumberFormat(NumberFormat newNumberFormat){ numberFormat =
   newNumberFormat; }
8.
9.  public char getCurrencySymbol(){ return currencySymbol; }
10. public NumberFormat getNumberFormat(){ return numberFormat; }
11. }
```

Листинг А. 164. PhoneNumber.java

```
1. public class PhoneNumber {
2.   private static String selectedInterPrefix;
3.   private String internationalPrefix;
4.   private String areaNumber;
5.   private String netNumber;
6.
7.   public PhoneNumber(String intPrefix, String areaNumber, String netNumber) {
8.      this.internationalPrefix = intPrefix;
9.      this.areaNumber = areaNumber;
10.     this.netNumber = netNumber;
11. }
12.
13.  public String getInternationalPrefix(){ return internationalPrefix; }
14.  public String getAreaNumber(){ return areaNumber; }
15.  public String getNetNumber(){ return netNumber; }
16.  public static String getSelectedInterPrefix(){ return selectedInterPrefix;
   }
17.
18.  public void setInternationalPrefix(String newPrefix){ internationalPrefix =
   newPrefix; }
19.  public void setAreaNumber(String newAreaNumber){ areaNumber =
   newAreaNumber; }
20.  public void setNetNumber(String newNetNumber){ netNumber = newNetNumber; }
21.  public static void setSelectedInterPrefix(String prefix){
   selectedInterPrefix = prefix; }
22.
23.  public String toString(){
24.     return internationalPrefix + areaNumber + netNumber;
25. }
26.}
```

Листинг А. 165. InternationalizedText.java

```
1. import java.util.Properties;
2. import java.io.File;
3. import java.io.IOException;
4. import java.io.FileInputStream;
5. public class InternationalizedText{
6.   private static final String DEFAULT_FILE_NAME = "";
7.   private Properties textProperties = new Properties();
8.
9.   public InternationalizedText(){
10.     this(DEFAULT_FILE_NAME);
11. }
12.  public InternationalizedText(String fileName){
13.     loadProperties(fileName);
14.  }
15.
16.  public void setFileName(String newFileName){
17.     if (newFileName != null){
18.       loadProperties(fileName);
19.     }
20.  }
21.  public String getProperty(String key){
22.   return getProperty(key, "" ) ;
23. }
24.  public String getProperty(String key, String defaultValue){
25.    return textProperties.getProperty(key, defaultValue);
```

```
26. }
27.
28. private void loadProperties(String fileName){
29.   try{
30.     FileInputStream input = new FileInputStream(fileName);
31.     textProperties.load(input) ;
32.   }
33.   catch (IOException exc) {
34.     textProperties = new Properties();
35.   }
36. }
37.}
```

Общие данные о форматах, принятых в стране, сохраняются в атрибутах вспомогательного класса Nation (листинг А.166). При первом создании экземпляра класса InternationalizationWizardпоследний создает коллекцию объектов Nation.

Листинг А.166. Nation.java

```
1. import java.text.NumberFormat;
2. public class Nation (
3.   private char symbol;
4.   private String name;
5.   private String dialingPrefix;
6.   private String propertyFileName;
7.   private NumberFormat numberFormat;
8.
9.   public Nation (String newName, char newSymbol, String newDialingPrefix,
10.    String newPropertyFileName, NumberFormat newNumberFormat) {
11.    name = newName;
12.    symbol = newSymbol;
13.    dialingPrefix = newDialingPrefix;
14.    propertyFileName = newPropertyFileName;
15.    numberFormat = newNumberFormat;
16.  }
17.
18.  public String getName() { return name; }
19.  public char getSymbol() ( return symbol; }
20.  public String getDialingPrefix(){ return dialingPrefix; }
21.  public String getPropertyFileName(){ return propertyFileName; }
22.  public NumberFormat getNumberFormat(){ return numberFormat; }
23.
24.  public String toString() { return name; }
25.}
```

Используем класс FacadeGui (листинг А.167) для демонстрации особенностей применения данного шаблона с точки зрения пользователя. Этот класс создает простой графический пользовательский интерфейс, с помощью которого можно наглядно представить эффект изменения страны, вызывая get-методы объекта класса InternationalizationWizardдля настройки языка, валюты и формата телефонных номеров.

Листинг А.167. FacadeGui.java

```
1. import java.awt.Container;
2. import java.awt.GridLayout;
3. import java.awt.event.ActionListener;
4. import java.awt.event.ActionEvent;
5. import java.awt.event.ItemListener;
6. import java.awt.event.ItemEvent;
7. import java.awt.event.WindowAdapter;
8. import java.awt.event.WindowEvent;
9. import javax.swing.BoxLayout;
10.import javax.swing.JButton;
11.import javax.swing.JComboBox;
12.import javax.swing.JFrame;
13.import javax.swing.JLabel;
14.import javax.swing.JPanel;
15.import javax.swing.JTextField;
16.public class FacadeGui implements ActionListener, ItemListener{
17.  private static final String GUI_TITLE = "title";
18.  private static final String EXIT_CAPTION = "exit";
19.  private static final String COUNTRY_LABEL = "country";
20.  private static final String CURRENCY_LABEL = "currency";
21.  private static final String PHONE_LABEL = "phone";
22.
23.  private JFrame mainFrame;
24.  private JButton exit;
25.  private JComboBox countryChooser;
26.  private JPanel controlPanel, displayPanel;
27.  private JLabel countryLabel, currencyLabel, phoneLabel;
28.  private JTextField currencyTextField, phoneTextField;
29.  private InternationalizationWizard nationalityFacade;
30.
31.  public FacadeGui(InternationalizationWizard wizard){
32.     nationalityFacade = wizard;
33.  }
34.
35.  public void createGui(){
36.     mainFrame = new JFrame(nationalityFacade.getProperty(GUI_TITLE));
37.     Container content = mainFrame.getContentPane();
38.     content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
39.
40.     displayPanel = new JPanel();
41.     displayPanel.setLayout(new GridLayout(3, 2));
42.
43.     countryLabel = new JLabel(nationalityFacade.getProperty(COUNTRY_LABEL));
44.     countryChooser = new JComboBox(nationalityFacade.getNations());
45.     currencyLabel = new
   JLabel(nationalityFacade.getProperty(CURRENCY_LABEL));
46.     currencyTextField = new JTextField();
47.     phoneLabel = new JLabel(nationalityFacade.getProperty(PHONE_LABEL));
48.     phoneTextField = new JTextField();
49.
50.     currencyTextField.setEditable(false);
51.     phoneTextField.setEditable(false);
52.
53.     displayPanel.add(countryLabel);
54.     displayPanel.add(countryChooser);
55.     displayPanel.add(currencyLabel);
56.     displayPanel.add(currencyTextField);
57.     displayPanel.add(phoneLabel);
58.     displayPanel.add(phoneTextField);
59.     content.add(displayPanel);
```

```
60.
61.     controlPanel = new JPanel();
62.     exit = new JButton(nationalityFacade.getProperty(EXIT_CAPTION));
63.     controlPanel.add(exit);
64.     content.add(controlPanel);
65.
66.     exit.addActionListener(this);
67.     countryChooser.addItemListener(this);
68.
69.     mainFrame.addWindowListener(new WindowCloseManager());
70.     mainFrame.pack();
71.     mainFrame.setVisible(true);
72. }
73.
74.  private void updateGui(){
75.     nationalityFacade.setNation(countryChooser.getSelectedItem().toString())
   ;
76.     mainFrame.setTitle(nationalityFacade.getProperty(GUI_TITLE));
77.     countryLabel.setText(nationalityFacade.getProperty(COUNTRY_LABEL));
78.     currencyLabel.setText(nationalityFacade.getProperty(CURRENCY_LABEL));
79.     phoneLabel.setText(nationalityFacade.getProperty(PHONE_LABEL));
80.     exit.setText(nationalityFacade.getProperty(EXIT_CAPTION));
81.
82.     currencyTextField.setText(nationalityFacade.getCurrencySymbol() + " " +
83.       nationalityFacade.getNumberFormat().format(5280.50));
84.    phoneTextField.setText(nationalityFacade.getPhonePrefix());
85.
86.     mainFrame.invalidate();
87.     countryLabel.invalidate();
88.     currencyLabel.invalidate();
89.     phoneLabel.invalidate();
90.     exit.invalidate();
91.     mainFrame.validate();
92. }
93.
94.  public void actionPerformed(ActionEvent evt){
95.     Object originator = evt.getSource();
96.     if (originator == exit){
97.       exitApplication();
98.     }
99.  }
100. public void itemStateChanged(ItemEvent evt){
101.    Object originator = evt.getSource();
102.    if (originator == countryChooser){
103.     updateGui();
104.    }
105. }
106.
107. public void setNation(Nation nation){
108.    countryChooser.setSelectedItem(nation);
109. }
110.
111. private class WindowCloseManager extends WindowAdapter{
112.    public void windowClosing(WindowEvent evt){
113.      exitApplication();
114.    }
115. }
116.
117. private void exitApplication(){
118.   System.exit(0);
119. }
120.}
```

Класс DataCreator (листинг А. 168) генерирует используемый в данном примере набор объектов класса InternationalizedText.

Листинг А. 168. DataCreator.java

```
1. import java.util.Properties;
2. import java.io.IOException;
3. import java.io.FileOutputStream;
4. public class DataCreator{
5.   private static final String GUI_TITLE = "title";
6.   private static final String EXIT_CAPTION = "exit";
7.   private static final String COUNTRY_LABEL = "country";
8.   private static final String CURRENCY_LABEL = "currency";
9.   private static final String PHONE_LABEL = "phone";
10.
11.  public static void serialize(String fileName){
12.    saveFrData ();
13.    saveUsData ();
14.    saveNlData ();
15. }
16.
17.  private static void saveFrData(){
18.    try{
19.      Properties textSettings = new Properties ();
20.      textSettings.setProperty(GUI_TITLE, "Demonstration du Pattern Facade");
21.      textSettings.setProperty(EXIT_CAPTION, "Sortir");
22.      textSettings.setProperty(COUNTRY_LABEL, "Pays");
23.      textSettings.setProperty(CURRENCY_LABEL, "Monnaie");
24.      textSettings.setProperty(PHONE_LABEL, "Numero de Telephone");
25.      textSettings.store(new FileOutputStream("french.properties"),
   "French Settings");
26.    }
27.    catch (IOException exc){
28.      System.err.println("Error storing settings to output");
29.      exc.printStackTrace();
30.    }
31. }
32.  private static void saveUsData(){
33.    try{
34.      Properties textSettings = new Properties ();
35.      textSettings.setProperty(GUI_TITLE, "Facade Pattern Demonstration");
36.      textSettings.setProperty(EXIT_CAPTION, "Exit");
37.      textSettings.setProperty(COUNTRY_LABEL, "Country");
38.      textSettings.setProperty(CURRENCY_LABEL, "Currency");
39.      textSettings. setProperty (PHONE_LABEL, "Phone Number");
40.      textSettings.store(new FileOutputStream("us.properties"), "US Settings");
41.    }
42.    catch (IOException exc){
43.      System.err.println("Error storing settings to output");
44.      exc.printStackTrace();
45.    }
46. }
47.  private static void saveNlData(){
48.    try{
49.      Properties textSettings = new Properties();
50.      textSettings.setProperty(GUI_TITLE, "Facade Pattern voorbeeld");
51.      textSettings.setProperty(EXIT_CAPTION, "Exit");
52.      textSettings.setProperty(COUNTRY_LABEL, "Land");
53.      textSettings.setProperty(CURRENCY_LABEL, "Munt eenheid");
54 . textSettings . setProperty (PHONE_LABEL, "Telefoonnummer") ;
```

```
55 .      textSettings.store(new FileOutputStream("dutch.properties"),
   "Dutch Settings");
56.     }
57.     catch (IOException exc){
58.      System.err.println("Error storing settings to output");
59.      exc.printStackTrace();
60.     }
61. }
62.}
```

Класс RunPattern, представленный в листинге А.169, создает экземпляр класса InternationalizationWizardи связывает его с графическим пользовательским интерфейсом. После этого можно использовать полученный объект, реализующий шаблон Facade, для представления информации о выбранной на данный момент стране.

Листинг А.169. RunPattern.java

```
1. import java.io.File;
2. public class RunPattern{
3.  public static void main(String [] arguments){
4.     System.out.println("Example for the Facade pattern");
5.     System.out.println() ;
6.     System.out.println("This code sample uses an InternationalizationWizard
   (a Facade)");
7.     System.out.println(" to manage communication between the rest of the
   application and");
8.     System.out.println(" a series of other classes.");
9.      System.out.println();
10.     System.out.println("The InternationalizationWizard maintains a
   collection of Nation") ;
11.     System.out.println(" objects. When the setNation method is called,
   the wizard sets the");
12.     System.out.println(" default nation, updating the Currency, PhoneNumber
   and localized");
13.     System.out.println(" String resources (InternationalizedText) available.");
14.     System.out.println ();
15.     System.out.println("Calls to get Strings for the GUI, the currency
   symbol or the dialing");
16.     System.out.println(" prefix are routed through the Facade, the
   InternationalizationWizard.");
17.     System.out.println ();
18.
19.     if (!(new File("data.ser").exists())) {
20 .      DataCreator.serialize("data.ser");
21.     }
22.
23.     System.out.println("Creating the InternationalizationWizard and setting
   the nation to US.");
24.     System.out.println();
25.     InternationalizationWizard wizard = new InternationalizationWizard();
26.     wizard.setNation("US");
27.
28.     System.out.println("Creating the FacadeGui.");
29.    System. out .println();
30 .     FacadeGui application = new FacadeGui(wizard);
31.     application.createGui();
32.     application.setNation(wizard.getNation("US"));
33. )
34.}
```

Flyweight

В данном примере шаблон Flyweight применяется для организации совместного использования объектов-состояний PIM-приложения. В примере, который был посвящен шаблону State, подобные объекты использовались для редактирования и хранения информации для набора объектов класса Appointment. В настоящем примере объекты-состояния задействованы для управления командами редактирования и сохранения нескольких коллекций объектов.

Интерфейс State (листинг А. 170) определяет стандартное поведение всех объектов-состояний приложения. В этом интерфейсе определены два базовых метода edit и save.

Листинг А. 170. State. *Java.*

```
1. package flyweight.example;
2.
3. import java.io.File
4. import java.io.IOException;
5. import java.io.Serializable;
6.
7. public interface State (
8.   public void save(File f, Serializable s) throws IOException;
9.   public void edit();
10.}
```

Интерфейс реализуется двумя классами: CleanState (листинг А.171) и DirtyState (листинг А. 172). В рассматриваемом примере данные классы применяются для отслеживания состояния нескольких объектов, в связи с чем необходим вспомогательный класс, который бы управлял тем, какой элемент нуждается в обновлении.

Листинг А. 171. CleanState.java

```
1. import java.io.File;
2. import java.io.FileOutputStream;
3. import java.io.IOException;
4. import java.io.ObjectOutputStream;
5. import java.io.Serializable;
6.
7. public class CleanState implements State{
8.   public void save(File file, Serializable s, int type) throws IOException{ )
9.
10.  public void edit(int type){
11.     StateFactory.setCurrentState(StateFactory.DIRTY);
12.     ((DirtyState)StateFactory.DIRTY).incrementStateValue(type);
13.  }
14.)
```

Листинг A.172. DirtyState.java

```
1. package flyweight.example;
2.
3. import java.io.File;
4. import.java.io.FileOutputStream;
5. import.java.io.IOException;
6. import.java.io.ObjectOutputStream;
7. import.java.io.Serializable;
8.
9. public class DirtyState implements State {
10.  public void save(File file, Serializable s) throws IOException {
11.     //сериализация
12.     FileOutputStream fos = new FileOutputStream(file);
13.     ObjectOutputStream out = new ObjectOutputStream(fos);
14.     out.writeObject(s);
15.  }
16.
17.  public void edit() {
18.     //опущено
19.  }
20.}
```

Поскольку эти два класса применяются для отслеживания общего состояния приложения, ими управляет класс StateFactory (листинг A.173), который по мере необходимости создает объекты обоих классов.

Листинг A. 173. StateFactory.java

```
1. public class StateFactory {
2.   public static final State CLEAN = new CleanState();
3.   public static final State DIRTY = new DirtyState ();
4.   private static State currentState = CLEAN;
5.
6.   public static State getCurrentState(){
7.      return currentState;
8.   }
9.
10.  public static void setCurrentState(State state){
11.     currentState = state;
12.  }
13.}
```

В рассматриваемом примере используется коллекция элементов, хранение которых обеспечивается классом ManagedList (листинг A.174). Применение этого класса позволяет гарантировать, что в заданном списке будут храниться лишь объекты определенного класса.

Листинг A.174. ManagedList.java

```
1. import java.util.ArrayList;
2. public class ManagedList{
3.   private ArrayList elements = new ArrayList();
4.   private Class classType;
5.
```

```
6.  public ManagedList(){ }
7.  public ManagedList(Class newClassType){
8.    classType = newClassType;
9.  }
10.
11. public void setClassType(Class newClassType){
12.   classType = newClassType;
13. }
14.
15. public void addItem(Object item){
16.   if ((item != null) && (classType.isInstance(item))){
17.     elements.add(item);
18.   } else {
19.     elements.add(item);
20.   }
21. }
22.
23. public void removeItem(Object item))
24.   elements.remove(item);
25. }
26.
27. public ArrayList getItems(){
28.   return elements;
29. }
30.}
```

Используемые в данном примере вспомогательные классы представляют прикладные объекты контактов, хранящихся в адресной книге, и соответствующих им адресов. Поведение этих объектов определяется интерфейсами Address (листинг А. 175) и Contact (листинг А. 177), а реализация этого поведения обеспечивается классами Addresslmpl (листинг А. 176) и Contactlmpl (листинг А. 178).

Листинг А. 175. Address.java

```
1. import java.io.Serializable;
2. public interface Address extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.   public static final String COMMA = ",";
6.   public String getType();
7.   public String getDescription() ;
8.   public String getStreet () ;
9.   public String getCityO;
10.  public String getState();
11.  public String getZipCode();
12.
13.  public void setType(String newType);
14.  public void setDescription(String newDescription) ;
15.  public void setStreet(String newStreet);
16.  public void setCity(String newCity);
17.  public void setState(String newState);
18.  public void setZipCode(String newZip);
19.}
```

Листинг A.176. AddressImpl.java

```
1. public class Addressimpl implements Address{
2.   private String type;
3.   private String description;
4.   private String street;
5.   private String city;
6.   private String state;
7.   private String zipCode;
8.   public static final String HOME = "home";
9.   public static final String WORK = "work";
10.
11.  public AddressImpl(){ }
12.  public AddressImpl(String newDescription, String newStreet,
13.     String newCity, String newState, String newZipCode){
14.       description = newDescription;
15.       street = newStreet;
16.       city = newCity;
17.       state = newState;
18.       zipCode = newZipCode;
19. }
20.
21.  public String getType(){ return type; }
22.  public String getDescription(){ return description; }
23.  public String getStreet(){ return street; }
24.  public String getCity(){ return city; }
25.  public String getstate(){ return state; }
26.  public String getZipCode(){ return zipCode; }
27.
28.  public void setType(String newType){ type = newType; )
29.  public void setDescription(String newDescription){ description =
     newDescription;}
30.  public void setStreet(String newStreet){ street = newStreet; }
31.  public void setCity(String newCity){ city = newCity; )
32.  public void setState(String newState){ state = newState; }
33.  public void setZipCode(String newZip){ zipCode = newZip; }
34.
35.  public String toString(){
36.     return street + EOL_STRING + city + COMMA + SPACE +
37.       state + SPACE + zipCode + EOL_STRING;
38. }
39.}
```

Листинг A. 177. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.    public String getFirstName();
5.    public String getLastName();
6.    public String getTitle();
7.    public String getOrganization();
8.
9.    public void setFirstName(String newFirstName);
10.   public void setLastName(String newLastName);
11.   public void setTitle(String newTitle);
12.   public void setOrganization(String newOrganization) ;
13.}
```

Листинг А.178. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization){
10.      firstName = newFirstName;
11.      lastName = newLastName;
12.      title = newTitle;
13.      organization = newOrganization;
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName(){ return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle(String newTitle){ title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization;}
25.
26. public String toString(){
27.    return firstName + SPACE + lastName;
28. }
29.}
```

Проверить реализацию данного шаблона на практике позволяет класс RunPattern (листинг А.179). Этот класс создает объекты класса ManagedList для хранения адресов и контактов, а затем использует общие объекты состояний для управления сохранением объектов в двух разных файлах.

Листинг А.179. RunPattern.java

```
1. public class RunPattern{
2.   public static void main(String [] arguments) throws java.io.IOException{
3.     System.out.println("Example for the Flyweight pattern");
4.      System.out.println();
5.     System.out.println("In this sample, State objects are shared between
    multiple");
6.     System.out.println(" parts of the PIM. Two lists, representing a Contact
    list");
7.     System.out.println(" and an Address Book, are used for the demonstration.");
8.      System.out.println(" The State objects - CleanState and DirtyState -
    represent");
9.     System.out.println(" the Flyweight objects in this example.");
10.     System.out.println();
11.
12.     System.out.println("Creating ManagedList objects to hold Contacts and
    Addresses");
13.     ManagedList contactList = new ManagedList(Contact.class);
14.     ManagedList addressList = new ManagedList(Address.class);
```

```
15.    System.out.println() ;
16.
17.    System.out.println("Printing the State for the application");
18.    printPIMState();
19.    System.out.println();
20.
21.    System.out.println("Editing the Address and Contact lists");
22.    StateFactory.getCurrentState().edit(State.CONTACTS);
23.    StateFactory.getCurrentState().edit(State.ADDRESSES) ;
24.    contactList.addItem(new ContactImpl("f", "1", "t", "o"));
25.    addressList.addItem(new AddressImpl("d", "s", "c", "st", "z"));
26.    System.out.println("Printing the State for the application");
27.    printPIMState ();
28.    System.out.println();
29.
30.    System.out.println("Saving the Contact list");
31.    StateFactory.getCurrentState() .save(new java.io.File("contacts.ser"),
   contactList.
32.    getItems(), State.CONTACTS);
33.    System.out.println("Printing the State for the application");
34.    printPIMState();
35.    System.out.println ();
36.
37.    System.out.println("Saving the Address list");
38.    StateFactory.getCurrentState().save(new java.io.File("addresses.ser"),
39.    addressList.getItems(), State.ADDRESSES);
40.    System.out.println("Printing the State for the application");
41.    printPIMState ();
42.  )
43.
44.  private static void printPIMState(){
45.    System.out.println(" Current State of the PIM: " +
   StateFactory.getCurrentState().
46.    getClassO);
47.    System.out.println(" Object ID: " +
   StateFactory.getCurrentState().hashCode());
48.    System.out.println() ;
49.  )
50.}
```

Half-Object Plus Protocol (HOPP)

PIM-приложение должно быть доступно с любого рабочего места, но его данные должны храниться централизованно. В данном примере используется шаблон HOPP и технология RMI, обеспечивающие хранения личного календаря на сервере с одновременной поддержкой возможности обращения удаленных пользователей к информации календаря.

Интерфейс Calendar (листинг А.180) определяет все методы, которые должны быть доступны в качестве удаленных. Этот интерфейс расширяет интерфейс `java.rmi.Remote`, а все его методы генерируют особые ситуации класса `java.rmi.RemoteException`. В данном примере интерфейс Calendar определяет три метода: getHost, `getAppointments` и `addAppointment`.

Листинг А. 180. Calendar.java

```
1. import java.rmi.Remote;
2. import.java.rmi.RemoteException;
3. import.java.util.Date;
4. import java.util.ArrayList;
5. public interface Calendar extends Remote{
6.   public String getHost() throws RemoteException;
7.   public ArrayList getAppointments(Date date) throws RemoteException;
8.   public void addAppointment(Appointment appointment, Date date) throws
   RemoteException;
9. }
```

Интерфейс Calendar реализуется двумя классами: удаленным RMI-объектом и его слепком, или прокси-объектом (см. раздел "Proxy" на стр. 483). Класс удаленного объекта CalendarImpl (листинг А.181) содержит реализацию методов, а слепок управляет взаимодействием с удаленным объектом. Для генерации слепка и опорного класса (skeleton class) нужно воспользоваться командой запуска компилятора RMI Java (rmic), указав ей в качестве параметра имя класса CalendarImpl. Опорный класс генерируется для обеспечения обратной совместимости, но начиная с Java 1.2 необходимость в нем уже не возникает.

Листинг А. 181. CalendarImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. import java.io.File;
4. import java.util.Date;
5. import java.util.ArrayList;
6. import java.util.HashMap;
7. public class CalendarImpl implements Calendar{
8.   private static final String REMOTE SERVICE = "calendarimpl";
9.   private static final String DEFAULT FILE NAME = "calendar.ser";
10.  private HashMap appointmentCalendar = new HashMap();
11.
12.  public CalendarImpl(){
13.    this(DEFAULT FILE NAME);
14.  }
15.  public CalendarImpl(String filename){
16.    File inputFile = new File(filename);
17.    appointmentCalendar = (HashMap)FileLoader.loadData(inputFile);
18.    if (appointmentCalendar == null){
19.     appointmentCalendar = new HashMap ();
20.   }
21.  try {
22.     UnicastRemoteObject.exportObject(this);
23.     Naming.rebind(REMOTE SERVICE, this);
24.   }
25.   catch (Exception exc) {
26.     System.err.println("Error using RMT to register the CalendarImpl " +
   exc);
27.   }
28. )
29.
30. public String getHost(){ return ""; }
31. public ArrayList getAppointments(Date date){
32.   ArrayList returnValue = null;
```

```
33.    Long appointmentKey = new Long(date.getTime());
34.    if (appointmentCalendar.containsKey(appointmentKey)){
35.      returnValue = (ArrayList)appointmentCalendar.get(appointmentKey);
36.    }
37.    return returnValue;
38.  }
39.
40.  public void addAppointment(Appointment appointment, Date date){
41.    Long appointmentKey = new Long(date.getTime());
42.    if (appointmentCalendar.containsKey(appointmentKey)){
43.      ArrayList appointments =
   (ArrayList)appointmentCalendar.get(appointmentKey);
44.        appointments.add(appointment);
45.      }
46.      else {
47.        ArrayList appointments = new ArrayList();
48.        appointments.add(appointment);
49.        appointmentCalendar.put(appointmentKey, appointments);
•50.     }
51.  }
52.}
```

Для того чтобы иметь возможность обрабатывать входящие коммуникационные запросы, объект класса CalendarImpl должен использовать вспомогательный класс RMI UnicastRemoteObject. В рассматриваемом примере конструктор класса Calendarlmpl экспортирует сам себя, используя статический метод UnicastRemoteObject .exportObject.

Кроме того, класс Calendarlmpl каким-то образом должен обеспечить доступ к себе извне. Для этого служба имен RMI вызывает rmiregistry. Эта команда должна быть выполнена до того, как будет создан объект класса Calendarlmpl. Работа команды rmiregistry строится по принципу телефонного справочника, поскольку она обеспечивает связь между объектом и его именем. Когда объект Calendarlmpl регистрирует себя с помощью команды rmiregistry, вызывая метод rebind, последний связывает имя "calendarimpl" со слепком соответствующего удаленного объекта.

Для того чтобы клиент смог использовать удаленный объект, ему нужно выполнить поиск в реестре RMI соответствующего узла сети и получить слепок нужного ему объекта. Слепок в данном случае можно сравнить с номером телефона — этот номер можно использовать из любой точки, воспользовавшись любым телефонным аппаратом, а после установления связи вы будете общаться с любым человеком, который возьмет трубку. В настоящем примере в качестве клиента объекта Calendarlmpl выступает класс CalendarHOPP (листинг А.182).

Листинг А.182. CalendarHOPP.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. import java.util.Date;
4. import java.util.ArrayList;
5. public class CalendarHOPP implements Calendar, java.io.Serializable{
6.   private static final String PROTOCOL = "rmi://";
7.   private static final String REMOTE SERVICE = "/calendarimpl";
8.   private static final String HOPP SERVICE = "calendar";
9.   private static final String DEFAULT HOST = localhost";
10.  private Calendar calendar;
```

```
11. private String host;
12.
13. public CalendarHOPP(){
14.    this(DEFAULT HOST);
15. }
16. public CalendarHOPP(String host) {
17.   try {
18.      this.host = host;
19.      String url = PROTOCOL + host + REMONE_SERVICE;
20.      calendar =  (Calendar)Naming.lookup(url);
21.      Naming.rebind(HOPP_SERVICE,  this):
22.    }
23.    catch  (Exception exc){
24.     System.err.println("Error using RMI to look up the CalendarImpl or
   register the CalendarHOPP " + exc);
25.    )
26.  }
27.  public String getHost(){ return host: ]
28.  public ArrayList getAppointments(Date date)  throws RemoteException{ return
   calendar.getAppointments(date); }
29.
30.
31.  public void addAppointment(Appointment appointment, Date date)  throws
   RemoteException { calendar.addAppointment(appointment, date); )
32.}
```

Класс CalendarHOPP имеет одно существенное отличие от обычного клиента RMI — он может запускать локально методы, которые в общем случае предназначаются для удаленного запуска. Это может оказаться весьма важным преимуществом с точки зрения снижения накладных расходов при обмене данными. Этот класс реализует тот же удаленный интерфейс Calendar, но не экспортирует себя. Он содержит ссылку на слепок и перенаправляет последнему все вызовы методов, которые данный класс не может обработать или не должен обрабатывать. Обеспечив такую пересылку, разработчик может реализовать в рассматриваемом классе те методы, которые, по мнению разработчика, должны выполняться локально. В данном примере к таким методам относится метод getHost. Объект данного класса можно зарегистрировать в реестре RMI подобно тому, как регистрируется обычный слепок. Единственное отличие заключается в том, что данный класс может выполнять методы локально.

Информацию, необходимую классу `Appointment` (листинг А.183), предоставляют вспомогательные классы, основанные на интерфейсах Contact (листинг А.184) и Location (листинг А.186). Реализация этих интерфейсов представлена классами `ContactImpl` (листинг А. 185) и `LocationImpl` (листинг А.187).

Листинг А. 183. Appointment.java

```
1. import java.io.Serializable;
2. import java.util.Date;
3. import java.util.ArrayList;
4. public class Appointment implements Serializable{
5.  private String description;
6.  private ArrayList contacts;
7.  private Location location;
8.  private Date startDate;
9.  private Date endDate;
10.
```

```
11. public Appointment(String description, ArrayList contacts, Location
    location, Date startDate, Date endDate){
12.    this.description = description;
13.    this.contacts = contacts;
14.    this.location = location;
15.    this.startDate = startDate;
16.    this.endDate = endDate;
17. }
18.
19. public String getDescription(){ return description; }
20. public ArrayList getContacts(){ return contacts; }
21. public Location getLocation(){ return location; }
22. public Date getStartDate(){ return startDate; }
23. public Date getEndDate(){ return endDate; }
24.
25. public void setDescription(String description){ this.description =
    description; }
26. public void setContacts(ArrayList contacts){ this.contacts = contacts; }
27. public void setLocation(Location location)) this.location = location; }
28. public void setStartDate(Date startDate){ this.startDate = startDate; }
29. public void setEndDate(Date endDate){ this.endDate = endDate; }
30.
31. public String toStringOf
32.   return "Appointment:" + "\n Description: " + description +
33.     "\n Location: " + location + "\n Start: " +
34.     startDate + "\n End: " + endDate + "\n";
35. }
36.}
```

Листинг A. 184. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String SPACE = " ";
4.   public String getFirstName();
5.   public String getLastName();
6.   public String getTitle();
7.   public String getOrganization();
8.
9.   public void setFirstName(String newFirstName);
10.  public void setLastName(String newLastName);
11.  public void setTitle(String newTitle);
12.  public void setOrganization(String newOrganization);
13.}
```

Листинг A. 185. ContactImpl.java

```
1. public class ContactImpl implements Contact)
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.     String newTitle, String newOrganization))
10.      firstName = newFirstName;
11.      lastName = newLastName;
```

```
12.      title = newTitle;
13.      organization = newOrganization;
14.  }
15.
16.  public String getFirstName(){ return firstName; }
17.  public String getLastName(){ return lastName; }
18.  public String getTitle(){ return title; }
19.  public String getOrganization(){ return organization; }
20.
21.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
22.  public void setLastName(String newLastName){ lastNarae = newLastName; }
23.  public void setTitle(String newTitle){ title = newTitle; }
24.  public void setOrganization(String newOrganization){ organization =
     newOrganization; }
25.
26.  public String toString(){
27.     return firstName + SPACE + lastName;
28.  }
29.}
```

Листинг А.186. Location.java

```
1. import java.io.Serializable;
2. public interface Location extends Serializable{
3.   public String getLocation();
4.   public void setLocation(String newLocation);
5. }
```

Листинг А.187. LocationImpl.java

```
1. public class LocationImpl implements Location{
2.   private String location;
3.
4.   public LocationImpl(){ }
5.   public LocationImpl(String newLocation){
6.      location = newLocation;
7.   }
8.
9.   public String getLocation(){ return location; }
10.
11.  public void setLocation(String newLocation)) location = newLocation; }
12.
13.  public String toString(){ return location; }
14.}
```

Класс FileLoader (листинг А.188) содержит методы, обеспечивающие загрузку коллекции объектов класса Appointment из файла, а в случае необходимости и сохранение ее в файле.

Листинг А.188. FileLoader.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
```

```
5. import java.io.ObjectInputStream;
6. import java.io.ObjectOutputStream;
7. import java.io.Serializable;
8. public class FileLoader{
9.  public static Object loadData(File inputFile){
10.    Object returnValue = null;
11.    try{
12.     if (inputFile.exists()){
13.          if (inputFile.isFile()){
14.              ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(inputFile));
15.               returnValue = readIn.readObject();
16.               readIn.close();
17.          }
18.         else (
19.             System.err.println(inputFile + " is a directory.");
20.          }
21.     }
22.     else {
23.        System.err.println("File " + inputFile + " does not exist.");
24.     }
25.    }
26.    catch (ClassNotFoundException exc){
27.     exc.printStackTrace();
28.
29.    }
30.    catch (lOException exc) {
31.     exc.printStackTrace();
32.
33.    }
34.    return returnValue;
35. )
36.  public static void storeData(File outputFile, Serializable data){
37.    try{
38.     ObjectOutputStream writeOut = new ObjectOutputStream(new
   FileOutputStream(outputFile));
39.     writeOut.writeObject(data);
40.     writeOut.close();
41.    }
42.    catch (lOException exc)(
43.     exc.printStackTrace();
44.    }
45. }
46.}
```

Использование данного шаблона на практике демонстрируется классом RunPattern (листинг А.189). Этот класс сначала создает экземпляры классов CalendarHOPP и Calendarlmpl. С помощью первого объекта сначала выполняется локальный вызов (метод getHost), а затем осуществляется вызов, который передается для обработки второму объекту в виде удаленного вызова, поскольку коллекцией объектов запланированных событий управляет объект класса Calendarlmpl.

Листинг А.189. RunPattern.java

```
1. import java.util.Calendar;
2. import java.util.Date;
3. import java.util.ArrayList;
4. import java.io.IOException;
5. import java.rmi.RemoteException;
```

```
6. public class RunPattern{
7.   private static Calendar dateCreator = Calendar.getInstance();
8.   public static void main(String [] arguments) throws RemoteException{
9.     System.out.println("Example for the HOPP pattern");
10.    System.out.println();
11.    System.out.println("This example will use RMI to demonstrate the HOPP
   pattern.");
12.    System.out.println(" In the sample, there will be two objects created,
   CalendarImpl");
13.    System.out.println(" and CalendarHOPP. The CalendarImpl object provides
   the true");
14.    System.out.println(" server-side implementation, while the CalendarHOPP
   would be");
15.    System.out.println(" a client or middle-tier representative.
   The CalendarHOPP will");
16.    System.out.println(" provide some functionality, in this case supplying
   the hostname");
17.    System.out.println(" in response to the getHost method.");
18.    System.out.println();
19.    System.out.println("Note: This example runs the rmiregistry,
   CalendarHOPP and CalendarImpl");
20.    System.out.println(" on the same machine.");
21.    System.out.println ();
22.
23.    try{
24.     Process p1 = Runtime.getRuntime().exec("rmic CalendarImpl");
25.     Process p2 = Runtime.getRuntime().exec("rmic CalendarHOPP");
26.      p1.waitFor O ;
27.     p2.waitFor () ;
28.    }
29.    catch (IOException exc) {
30.      System.err.println("Unable to run rmic utility. Exiting
   application.");
31.      System.exit(1);
32.    }
33.    catch (InterruptedException exc) {
34.      System.err.println("Threading problems encountered while using the
   rmic utility.");
35.    }
36.
37.    System.out.println("Starting the rmiregistry");
38.    System.out.println();
39.    Process rmiProcess = null;
40.    try{
41.      rmiProcess = Runtime.getRuntime().exec("rmiregistry");
42.      Thread.sleep(15000);
43.    }
44.    catch (lOException exc){
45.      System.err.println("Unable to start the rmiregistry. Exiting
   application.");
46.      System.exit(l) ;
47.    }
48.    catch (InterruptedException exc){
49.      System.err.println("Threading problems encountered when starting the
   rmiregistry.");
50.    }
51.
52.    System.out.println("Creating the CalendarImpl object, which provides the
   server-side implementation.");
53.    System.out.println("(Note: If the CalendarImpl object does not have a
   file containing Appointments,");
54.    System.out.println(" this call will produce an error message. This will
   not affect the example.)");
```

```
55.    CalendarImpl remoteObject = new CalendarImpl();
56.
57.     System.out.println();
58.    System.out.println("Creating the CalendarHOPP object, which provides
   client-side functionality.");
59.    CalendarHOPP localObject = new CalendarHOPP();
60.
61.     System.out.println();
62.    System.out.println("Getting the hostname. The CalendarHOPP will handle
   this method locally.");
63.    System.out.println("Hostname is " + localObject.getHost());
64.    System.out.println ();
65.
66.    System.out.println("Creating and adding appointments. The CalendarHOPP
   will forward");
67.    System.out.println(" these calls to the CalendarImpl object.");
68.    Contact attendee = new ContactImpl("Jenny", "Yip", "Chief Java Expert",
   "MuchoJavaLTD");
69.    ArrayList contacts = new ArrayList();
70.     contacts.add(attendee);
71.    Location place = new LocationImpl("Albuquerque, NM");
72.    localObject.addAppointment(new Appointment("Opening speeches at annual
   Java Guru's dinner",
73.     contacts, place, createDate(2001, 4, 1, 16, 0),
74.     createDate(2001, 4, 1, 18, 0)), createDate(2001, 4, 1, 0, 0));
75.    localObject.addAppointment(new Appointment("Java Guru post-dinner Cafe
   time",
76.     contacts, place, createDate(2001, 4, 1, 19, 30),
77.    createDate(2001, 4, 1, 21, 45)), createDate(2001, 4, 1, 0, 0) ) ;
78.    System.out.println("Appointments added.");
79.     System.out.println();
80.
81.    System.out.println("Getting the Appointments for a date. The
   CalendarHOPP will forward");
82.    System.out.println(" this call to the CalendarImpl object.");
83.    System.out.println(localObject.getAppointments(createDate(2001, 4, 1, 0,
   0)));
84. >
85.
86.  public static Date createDate(int year, int month, int day, int hour, int
   minute){
87.    dateCreator.set(year, month, day, hour, minute);
88.     return dateCreator.getTime();
89. }
90.}
```

Proxy

Поскольку в адресной книге хранятся все сведения о профессиональных и личных контактах пользователя, со временем ее размер выходит за оптимальные пределы. Кроме того, пользователю далеко не в каждом сеансе работы с **PIM-приложением** нужна адресная книга. По этим причинам необходимо создать некий промежуточный объект адресной книги, который должен быстро загружаться при запуске приложения. Для реализации этой задачи в данном примере используется шаблон Proxy, в соответствии с которым создается прокси-объект, представляющий реальную адресную книгу.

В листинге А.190 представлен интерфейс AddressBook, с помощью которого обеспечивается доступ к адресной книге PIM-приложения. Этот интерфейс должен определять, как минимум, методы добавления новых контактов, а также получения и сохранения адресов.

Листинг А. 190. AddressBook.java

```
1. import java.io.IOException;
2. import java.util.AttayList;
3. public interface AddressBook {
4.  public void add(Address address);
5.  public ArrayList getAllAddresses () ;
6.  public Address getAddress(String description);
7.
8.  public void open();
9.  public void saved;
10.}
```

На получение данных из адресной книги может потребоваться весьма длительное время (именно этим, по-видимому, объясняется повсеместная нелюбовь пользователей к адресным книгам). Поэтому прокси-объект должен оттягивать, насколько это возможно, момент создания реальной адресной книги. Конечно, вся ответственность за создание адресной книги лежит именно на прокси-объекте AddressBookProxy (листинг А.191), но выполнить эту операцию он должен только тогда, когда она действительно нужна.

Листинг А.191. AddressBookProxy.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. public class AddressBookProxy implements AddressBook{
6.  private File file;
7.  private AddressBookImpl addressBook;
8.  private ArrayList localAddresses = new ArrayList ();
9.
10.  public AddressBookProxy(String filename){
11.     file = new File(filename);
12. }
13.
14.  public void open(){
15.     addressBook = new AddressBookImpl(file);
16.    Iteraror addressIterator = localAddresses.iterator() ;
17.     while (addressIterator.hasNext()){
18.       addressBook.add((Address)addressIterator.next() ) ;
19.     }
20. )
21.
22.  public void save(){
23.     if (addressBook != null){
24.       addressBook.save();
25.     } else if (!localAddresses.isEmpty()){
26.       open ();
27.       addressBook.save();
28.     }
```

```
29.  }
30.
31. public ArrayList getAllAddresses () {
32.    if (addressBook == null) {
33.     open () ;
34.    }
35.    return addressBook.getAllAddresses();
36.  }
37.
38.  public Address getAddress(String description){
39.     if (!localAddresses.isEmpty()){
40.       Iterator addressIterator = localAddresses.iterator();
41.       while (addressIterator.hasNext()){
42.            AddressImpl address = (AddressImpl)addressIterator.next();
43.            if (address.getDescription().equalsIgnoreCase(description)){
44.                 return address;
45.            }
46.       }
47.     }
48.     if (addressBook == null){
49.       open();
50.     }
51.     return addressBook.getAddress(description);
52. )
53.
54.  public void add(Address address){
55.     if (addressBook != null){
56.       addressBook.add(address);
57.    } else if (!localAddresses.contains(address) ) {.
58.       localAddresses.add(address);
59.     }
60. }
61.}
```

Обратите внимание на то, что класс AddressBookProxy имеет собственную коллекцию ArrayList, предназначенную для хранения адресов. Это позволяет прокси-объекту в тех случаях, когда пользователь добавляет адрес с помощью метода add, использовать свою внутреннюю адресную книгу, не обращаясь к реальной адресной книге.

Реальная адресная книга в приложении представлена классом AddressBookImpl (листинг А. 192). Этот класс связан с файлом, в котором сохраняется содержимое коллекции ArrayList, представляющее собой все адреса, когда-либо внесенные пользователем в адресную книгу. Экземпляр класса AddressBookProxy создает объект класса AddressBookImpl только в том случае, когда это действительно необходимо, например, когда пользователь вызывает метод `getAllAddresses`.

Листинг А.192. AddressBookImpl.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. public class AddressBookImpl implements AddressBook {
6.  private File file;
7.  private ArrayList addresses = new ArrayList ();
8.
9.  public AddressBookImpl(File newFile) {
10.    file = newFile;
11.    open();
```

```
12. }
13.
14. public ArrayList getAllAddresses(){ return addresses; }
15.
16. public Address getAddress(String description) {
17.    Iterator addressIterator = addresses.iterator();
18.    while (addressIterator.hasNext()){
19.     AddressImpl address = (AddressImpl)addressIterator.next();
20.     if (address.getDescription().equalsIgnoreCase(description)){
21.         return address;
22.     }
23.    }
24.    return null;
25. }
26.
27. public void add(Address address) {
28.    if (!addresses.contains(address)){
29.      addresses.add(address);
30.    }
31. }
32.
33. public void open(){
34.    addresses = (ArrayList)FileLoader.loadData(file);
35. }
36.
37. public void save(){
38.   FileLoader.storeData(file, addresses);
39. }
40.}
```

Класс `AddressBookImpl` делегирует задачу загрузки и сохранения файлов специализированному классу, названному `FileLoader` (листинг А.193). У этого класса имеются методы, позволяющие считывать из файла объекты, классы которых реализуют интерфейс `Serializable`, а также записывать такие объекты в файл.

Листинг А.193. FileLoader.java

```
1. import java.io.File;
2. import java.io.FileInputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. import java.io.ObjectInputStream;
6. import java.io.ObjectOutputStream;
7. import java.io.Serializable;
8. public class FileLoader{
9.   public static Object loadData(File inputFile){
10.    Object returnValue = null;
11.    try{
12.      if (inputFile.exists()){
13.          if (inputFile.isFile()){
14.              ObjectInputStream readIn = new ObjectInputStream(new
   FileInputStream(inputFile));
15.              returnValue = readIn.readObject();
16.              readIn.close();
17.          }else{
18.             System.err.println(inputFile + " is a directory.");
19.          >
20.      }else{
21.         System.err.println("File " + inputFile + " does not exist.");
22.      }
```

```
23.    }catch (ClassNotFoundException exc){
24.      exc.printStackTrace();
25.    }catch (IOException exc){
26.      exc.printStackTrace();
27.    }
28.    return returnValue;
29. )
30. public static void storeData(File outputFile, Serializable data){
31.    try{
32.      ObjectOutputStream writeOut = new ObjectOutputStream(new
   FileOutputStream(outputFile));
33.      writeOut.writeObject(data);
34.      writeOut.close();
35.    }catch (IOException exc){
36.      exc.printStackTrace();
37.    }
38. }
39.}
```

Интерфейс Address (листинг А.194) и реализующий его класс `AddressImpl` (листинг А.195) в данном примере используются для обеспечения хранения информации об адресах.

Листинг А.194. Address.java

```
1. import java.io.Serializable;
2. public interface Address extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.   public static final String COMMA = ",";
6.   public String getAddress () ;
7.   public String getTypeO;
8.   public String getDescription();
9.   public String getStreet();
10.  public String getCityO;
11.  public String getStateO;
12.  public String getZipCode();
13.
14.  public void setType(String newType);
15.  public void setDescription(String newDescription) ;
16.  public void setStreet(String newStreet);
17.  public void setCity(String newCity);
18.  public void setState(String newState);
19.  public void setZipCode(String newZip);
20.}
```

Листинг А.195. AddressImpl.java

```
1. public class AddressImpl implements Address{
2.   private String type;
3.   private String description;
4.   private String street;
5.   private String city;
6.   private String state;
7.   private String zipCode;
8.   public static final String HOME = "home";
```

```
9.  public static final String WORK = "work";
10.
11. public AddressImpl(){ }
12. public AddressImpl(String newDescription, String newStreet,
13.   String newCity, String newState, String newZipCode){
14.     description = newDescription;
15.     street = newStreet;
16.     city = newCity;
17.     state = newState;
18.     zipCode = newZipCode;
19. }
20.
21. public String getType(){ return type; }
22. public String getDescription (){ return description; }
23. public String getstreet(){ return street; }
24. public String getCity(){ return city; }
25. public String getstate(){ return state; }
26. public String getZipCode(){ return zipCode; }
27.
28. public void setType(String newType){ type = newType; }
29. public void setDescription(String newDescription){ description =
    newDescription; }
30. public void setStreet(String newStreet){ street = newStreet; }
31. public void setCity(String newCity)) city = newCity; }
32. public void setState(String newState){ state = newState; }
33. public void setZipCode(String newZip){ zipCode = newZip; }
34.
35. public String toString(){
36.    return description;
37. }
38. public String getAddress(){
39.    return description + EOL_STRING + street + EOL_STRING +
40.     city + COMMA + SPACE + state + SPACE + zipCode + EOL_STRING;
41. }
42.}
```

Класс DataCreator (листинг А. 196) создает тестовый файл, в котором хранится информация о нескольких адресах.

Листинг А. 196. DataCreator.java

```
1. import java.io.Serializable;
2. import java.io.ObjectOutputStream;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
5. import java.util.ArrayList;
6. public class DataCreator{
7.   private static final String DEFAULT_FILE = "data.ser";
8.
9.   public static void main(String [] args)(
10.     String fileName;
11.     if (args.length == 1){
12.       fileName = args[0];
13.     }else(
14.       fileName = DEFAULT_FILE;
15.     }
16.     serialize(fileName);
17.  }
18.
19.  public static void serialize(String fileName){
```

```
20.    try{
21.     serializeToFile(createData() , fileName);
22.    } catch (IOException exc)(
23.     exc.printStackTrace() ;
24.    }
25. }
26.
27. private static Serializable createData(){
28.    ArrayList items = new ArrayList();
29.    items.add(new AddressImpl("Home address", "1418 Appian Way",
   "Pleasantville", "NH", "27415"));
30.    items.add(new AddressImpl("Resort", "711 Casino Ave.", "Atlantic City",
   "NJ", "91720"));
31.    items.add(new AddressImpl("Vacation spot", "90 Ka'ahanau Cir.",
   "Haleiwa", "HI", "41720"));
32.    return items;
33. }
34.
35. private static void serializeToFile(Serializable data, String fileName)
   throws IOException{
36.    ObjectOutputStream serOut = new ObjectOutputStream(new
   FileOutputStream(fileName)) ;
37.    serOut.writeObject(data) ;
38.   serOut.close ();
39. }
40.}
```

Использование прокси-объекта на практике можно увидеть с помощью класса RunPattern, исходный код которого представлен в листинге А.197. Сначала он создает экземпляр класса AddressBookProxy и заносит в полученный прокси-объект несколько новых адресов, которые поначалу хранятся локально. И только при вызове метода getAllAddresses прокси-объект создает экземпляр класса AddressBookImpl и извлекает адреса, ранее сохраненные в файле.

Листинг А. 197. RunPattern.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. public class RunPattern{
5.  public static void main(String [] arguments){
6.     System.out.println("Example for the Proxy pattern");
7.      System.out.println();
8.     System.out.println("This code will demonstrate the use of a Proxy to");
9.     System.out.println(" provide functionality in place of its underlying");
10.    System.out.println(" class.");
11.    System.out.println() ;
12.
13.    System.out.println(" Initially, an AddressBookProxy object will
   provide");
14.    System.out.println(" address book support without requiring that the");
15.    System.out.println(" AddressBookImpl be created. This could
   potentially");
16.    System.out.println(" make the application run much faster, since the");
17.    System.out.println (" AddressBookImpl would need to read in all addresses");
18.    System.out.println(" from a file when it is first created.");
19.     System.out.println();
20.
21.    if (! (new File("data.ser") .exists ()))(
```

```
22.       DataCreator.serialize("data.ser");
23.     }
24.     System.out.println("Creating the AddressBookProxy");
25.     AddressBookProxy proxy = new AddressBookProxy("data.ser");
26.     System.out.println("Adding entries to the AddressBookProxy");
27.     System.out.println("(this operation can be done by the Proxy, without");
28.     System.out.println(" creating an AddressBookImpl object)");
29.     proxy.add(new AddressImpl("Sun Education [CO]", "500 El Dorado Blvd.",
  "Broomfield", "CO", "80020"));
30.     proxy.add(new AddressImpl("Apple Inc.", "1 Infinite Loop", "Redwood
  City", "CA", "93741"));
31.     System.out.println("Addresses created. Retrieving an address");
32.     System.out.println("(since the address is stored by the Proxy, there is");
33.     System.out.println(" still no need to create an AddressBookImpl object)");
34.     System.out.println();
35.     System.out.println(proxy.getAddress("Sun Education [CO]").getAddress());
36.     System.out.println();
37.
38.   System.out.println("So far, all operations have been handled by the Proxy,");
39.     System.out.println(" without any involvement from the
  AddressBookImpl.");
40.     System.out.println(" Now, a call to the method getAllAddresses will");
41.     System.out.println(" force instantiation of AddressBookImpl, and will");
42.     System.out.println(" retrieve ALL addresses that are stored.");
43.     System.out.println();
44.
45.     ArrayList addresses = proxy.getAllAddresses();
46.     System.out.println("Addresses retrieved. Addresses currently stored:");
47.     System.out.println(addresses) ;
48.   }
49. }
```

Системные шаблоны

Model-View-Controller (MVC)

В данном примере показана реализация шаблона MVC на компонентном уровне, обеспечивающая управление контактами в PIM-приложении. Класс `ContactModel` (листинг А. 198) в данном примере является моделью и содержит информацию об имени и фамилии контактного лица, а также об организации, которую оно представляет.

Листинг А. 198. `ContactModel.java`

```
1. import java.util.ArrayList;
2. import java.util.Iterator;
3. public class ContactModel{
4.   private String firstName;
5.   private String lastName;
6.   private String title;
7.   private String organization;
8.  private ArrayList contactViews = new ArrayList ();
9.
10. public ContactModel () (
11.     this(null);
12.  }
13.  public ContactModel  (ContactView view){
14.     firstName = "";
15.     lastName = "";
16.     title = "";
17.     organization = "";
18.     if  (view != null){
19.       contactViews.add(view);
20.     }
21.  }
22.
23.  public void addContactView(ContactView){
24.     if  (!contactViews.contains(view)){
25.       contactViews.add(view);
26.     }
27.  }
28.
29.  public void removeContactView(ContactView view) {
30.    contactViews.remove(view) ;
31.  }
32.
33.  public String getFirstName(){ return firstName; }
34.  public String getLastName(){ return lastName; )
35.  public String getTitle(){ return title; }
36.  public String getOrganization(){ return organization; }
37.
38.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
39.  public void setLastName(String newLastName){ lastName = newLastName; }
40.  public void setTitle(String newTitle){ title = newTitle; )
41.  public void setOrganization(String newOrganization){ organization =
   newOrganization; )
42.
43.  public void updateModel(String newFirstName, String newLastName,
44.     String newTitle, String newOrganization){
45.     if  (!isEmptyString(newFirstName)){
46.       setFirstName(newFirstName);
47.     }
48.     if  (!isEmptyString(newLastName)){
49.      setLastName(newLastName) ;
```

```
50.     }
51.     if (!isEmptyString(newTitle)){
52.       setTitle(newTitle);
53.     }
54.     if (!isEmptyString(newOrganization)){
55.       setOrganization(newOrganization);
56.     }
57.     updateView();
58. }
59.
60.  private boolean isEmptyString input){
61.     return ((input == null) II input.equals(""));
62. }
63.
64.  private void updateView(){
65.     iterator notifyViews = contactViews.iterator();
66.     while (notifyViews.hasNext()){
67.       ((ContactView)notifyViews.next()).refreshContactView(firstName,
   lastName, title, organization);
68.     }
69. )
70. }
```

Класс ContactModel содержит коллекцию ArrayList объектов класса ContactView (листинг А.199) и обновляет последние при каждом изменении данных модели. Стандартное поведение всех представлений определяется методом refreshContactView интерфейса ContactView.

Листинг А. 199. ContactView.java

```
1. public interface ContactView{
2.   public void refreshContactView(String firstName,
3.     String lastName, String title, String organization);
4. }
```

В данном примере используется два представления. Первое, ContactDisplayView (листинг А.200), обеспечивает отображение обновленной информации модели, но не поддерживает работу с контроллером, являясь примером реализации поведения "только чтение".

Листинг А.200. ContactDisplayView.java

```
1. import javax.swing.JPanel;
2. import javax.swing.JScrollPane;
3. import javax.swing.JTextArea;
4. import java.awt.BorderLayout;
5. public class ContactDisplayView extends JPanel implements ContactView{
6.  private JTextArea display;
7.
8.  public ContactDisplayView(){
9.    createGui();
10.  }
11.
12. public void createGui(){
13.     setLayout(new BorderLayout());
14.    display = new JTextArea(10, 40);
15.    display.setEditable(false);
```

```
16.     JScrollPane scrollDisplay = new JScrollPane(display);
17.     this.add(scrollDisplay, BorderLayout.CENTER);
18.   }
19.
20.   public void refreshContactView(String newFirstName,
21.     String newLastName, String newTitle, String newOrganization)(
22.     display.setText("UPDATED CONTACT:\nNEW VALUES:\n" +
23.       "\tName: " + newFirstName + " " + newLastName +
24.       "\n" + "\tTitle: " + newTitle + "\n" +
25.       "\tOrganization: " + newOrganization);
26.   }
27.}
```

Второе представление, `ContactEditView` (листинг А.201), позволяет пользователю обновить контактную информацию, определенную моделью.

Листинг А.201. ContactEditView.java

```
1. import javax.swing.BoxLayout;
2. import javax.swing.JButton;
3. import javax.swing.JLabel;
4. import javax.swing.JTextField;
5. import javax.swing.JPanel;
6. import java.awt.GridLayout;
7. import java.awt.BorderLayout;
8. import java.awt.event.ActionListener;
9. import java.awt.event.ActionEvent;
10.public class ContactEditView extends JPanel implements ContactView{
11.  private static final String UPDATE_BUTON = "Update";
12.  private static final String EXIT_BUTON = "Exit";
13.  private static final String CONTACT_FIRST_NAME = "First Name ";
14.  private static final String CONTACT_LAST_NAME = "Last Name ";
15.  private static final String CONTACT_TITLE = "Title ";
16.  private static final String CONTACT_ORG = "Organization ";
17.  private static final int FNAME_COL_WIDTH = 25;
18.  private static final int LNAME_COL_WIDTH = 40;
19.  private static final int TITLE_COL_WIDTH = 25;
20.  private static final int ORG_COL_WIDTH = 40;
21.  private ContactEditController controller;
22.  private JLabel firstNameLabel, lastNameLabel, titleLabel,
   organizationLabel;
23.  private JTextField firstName, lastName, title, organization;
24.  private JButton update, exit;
25.
26.  public ContactEditView(ContactModel model){
27.     controller = new ContactEditController(model, this);
28.     createGui();
29.  }
30.  public ContactEditView(ContactModel model, ContactEditController
   newController){
31.     controller = newController;
32.     createGui();
33.  }
34.
35.  public void createGui(){
36.     update = new JButton(UPDATE_BUTTON);
37.     exit = new JButton(EXIT_BUTTON);
38.
39.     firstNameLabel = new JLabel(CONTACT_FIRST_NAME);
40.     lastNameLabel(CONTACT_LAST_NAME);
41.     titleLabel = =new JLabel(CONTACT_TITLE);
```

```
42.     organizationLabel = new JLabel(CONTACT_ORG);
43.
44.     firstName = new JTextField(FNAME_COL_WIDTH);
45.     lastName = new JTextField(LNAME_COL_WIDTH);
46.     title = new JTextField(TITLE_COL_WIDTH);
47.     organization = new JTextField(ORG_COL_WIDTH);
48.
49.     JPanel editPanel = new JPanelO;
50.
51.     editPanel.setLayout(new BoxLayout(editPanel, BoxLayout.X_AXIS));
52.     JPanel labelPanel - new JPanel();
53.     labelPanel.setLayout(newGridLayout(0, 1));
54.
55.     labelPanel.add(firstNameLabel);
56.     labelPanel.add(lastNameLabel);
57.     labelPanel.add(titleLabel);
58.     labelPanel.add(organizationLabel);
59.
60.     editPanel.add(labelPanel);
61.
62.     JPanel fieldPanel = new JPanel();
63.     fieldPanel.setLayout(newGridLayout(0, 1));
64.
65.     fieldPanel.add(firstName);
66.     fieldPanel.add(lastName);
67.     fieldPanel.add(title);
68.     fieldPanel.add(organization);
69.
70.     editPanel.add(fieldPanel);
71.
72.     JPanel controlPanel = new JPanelO;
73.     controlPanel.add(update);
74.     controlPanel.add(exit);
75.     update.addActionListener(controller);
76.     exit.addActionListener(new ExitHandler());
77.
78.     setLayout(new BorderLayout());
79.     add(editPanel, BorderLayout.CENTER);
80.     add(controlPanel, BorderLayout.SOUTH);
81.   }
82.
83.  public Object getUpdateRef(){ return update; }
84.  public String getFirstName(){ return firstName.getText(); }
85.  public String getLastName(){ return lastName.getText(); }
86.  public String getTitle(){ return title.getText(); }
87.  public String getOrganization(){ return organization.getText(); }
88.
89.  public void refreshContactView(String newFirstName,
90.     String newLastName, String newTitle,
91.     String newOrganization){
92.     firstName.setText(newFirstName);
93.     lastName.setText(newLastName);
94.     title.setText(newTitle);
95.     organization.setText(newOrganization);
96.   }
97.
98.  private class ExitHandler implements ActionListener{
99.     public void actionPerformed(ActionEvent event){
100.      System.exit(0);
101.    }
102. }
103.}
```

Обновление модели возможно благодаря наличию контроллера, связанного с классом ContactEditView. В данном примере управление взаимодействием между экземпляром класса ContactEditView и связанным с ним экземпляром класса ContactEditController (листинг А.202) осуществляется на основе средств обработки событий языка Java (с применением расширения шаблона Observer). Объект класса ContactEditController обновляет объект класса `ContactModel`, когда процесс обновления инициируется объектом класса ContactEditView. Для обновления вызывается метод `updateModel` с новыми данными, полученными из редактируемых полей связанного с экземпляром класса ContactEditController представления.

Листинг А.202. `ContactEditController.java`

```
1. import java.awt.event.*;
2.
3. public class ContactEditController implements ActionListener{
4.  private ContactModel model;
5.  private ContactEditView view;
6.
7.  public ContactEditController (ContactModel m, ContactEditView v) {
8.     model = m;
9.     view = v;
10. }
11.
12.  public void actionPerformed(ActionEvent evt){
13.    Object source = evt.getSource();
14.    if (source == view.getUpdateRef()){
15.      updateModel();
16.    }
17. }
18.
19.  private void updateModel(){
20.    String firstName = null;
21.    String lastName = null;
22.    if (isAlphabetic(view.getFirstName())){
23.    firstName = view.getFirstName();
24. }
25.  if (isAlphabetic(view.getLastName())){
26.    lastName = view.getLastName();
27. }
28. model.updateModel( firstName, lastName,
29.   view.getTitle() , view.getOrganization ());
30. }
31.
32.  private boolean isAlphabetic(String input){
33.   char [] testChars = {'1', '2', '3', '4', '5', '6','7', '8', '9', '0'};
34.    for (int i = 0; i < testChars.length; i++){
35.      if (input.indexOf(testChars[i]) != -1){
36.           return false;
37.      }
38.    }
39.    return true;
40. }
41.}
```

Класс RunPattern, исходный код которого представлен в листинге А.203, предназначен для иллюстрации практического применения данного шаблона. Этот класс создает модель и графические пользовательские интерфейсы для обоих представле-

ний, связанных с данной моделью. Обновляемая информация, которая поступает от экземпляра класса ContactEditView, отображается экземпляром класса ContactDisplayView, что наглядно демонстрирует, как одна и та же модель может предоставлять информацию нескольким объектам представлений.

Листинг A.203. RunPattern.java

```
1. import java.awt.Container;
2. import java.awt.event.WindowAdapter;
3. import java.awt.event.WindowEvent;
4. import javax.swing.JFrame;
5. import javax.swing.JPanel;
6. public class RunPattern{
7.  public static void main(String [] arguments){
8.     System.out.println("Example for the MVC pattern");
9.      System.out.println();
10.    System.out.println("In this example, a Contact is divided into");
11.    System.out.println(" Model, View and Controller components.");
12.     System.out.println();
13.    System.out.println("To illustrate the flexibility of MVC, the same");
14.    System.out.println (" Model will be used to provide information");
15.    System.out.println(" to two View components.");
16.     System.out.println();
17.    System.out.println("One view, ContactEditView, will provide a Contact");
18.    System.out.println(" editor window and will be paired with a controller");
19.    System.out.println(" called ContactEditController.");
20.     System.out.println();
21.    System.out.println("The other view, ContactDisplayView, will provide a");
22.    System.out.println (" display window which will reflect the changes made");
23.    System.out.println(" in the editor window. This view does not support");
24.   System.out.println(" user interaction, and so does not provide a controller.");
25.     System.out.println();
26.
27.     System.out.println("Creating ContactModel");
28.     ContactModel model = new ContactModel();
29.
30.     System.out.println("Creating ContactEditView and ContactEditController");
31.     ContactEditView editorView = new ContactEditView(model);
32  .   model.addContactView(editorView);
33.     createGui(editorView, "Contact Edit Window");
34.
35.     System.out.println("Creating ContactDisplayView");
36.     ContactDisplayView displayView = new ContactDisplayView();
37.     model.addContactView(displayView);
38.     createGui(displayView, "Contact Display Window");
39. }
40.
41.  private static void createGui(JPanel contents, String title))
42.    JFrame applicationFrame = new JFrame (title) ;
43.     applicationFrame.getContentPane() .add(contents);
44.     applicationFrame.addWindowListener(new WindowCloseManager());
45.     applicationFrame.pack();
46.     applicationFrame.setVisible(true) ;
47. )
48.
49.   private static class WindowCloseManager extends WindowAdapter{
50.      public void windowClosing(WindowEvent evt){
51.        System.exit(0);
52.      ;
53. }
54.}
```

Session

В данном примере клиент обращается к серверу с запросами на выполнение ряда операций по обновлению контактной информации в совместно используемой адресной книге. Пользователь в данном случае может выполнять четыре операции:

- добавление контакта;
- добавление адреса (связан с текущим контактом);
- удаление адреса (связан с текущим контактом);
- сохранение изменений контакта и его адреса.

Эти операции определяются в классе SessionClient (листинг А.204).

Листинг А.204. SessionClient.java

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. public class SessionClient{
6.   private static final String SESSION_SERVER_SERVICE_NAME = "sessionServer";
7.   private static final String SESSION_SERVER_MACHINE_NAME = "localhost";
8.   private long sessionID;
9.   private SessionServer sessionServer;
10.
11.  public SessionClient(){
12.   try{
13.       String url = "//" + SESSION SERVER MACHINE NAME + "/" +
14.SESSION_SERVER_SERVICE_NAME;
15.       sessionServer = (SessionServer)Naming.lookup(url);
16.    }
17.    catch (RemoteException exc){}
18.    catch (NotBoundException exc){}
19.    catch (MalformedURLException exc){}
20.    catch (ClassCastException exc){}
21.  }
22.
23.  public void addContact(Contact contact) throws SessionException{
24.   try{
25.      sessionID = sessionServer.addContact(contact, 0);
26.    }
27.    catch (RemoteException exc) {}
28.)
29.
30.  public void addAddress(Address address) throws SessionException{
31.   try{
32.      sessionServer.addAddress(address, sessionID);
33.    }
34.    catch (RemoteException exc){}
35.  }
36.
37.  public void removeAddress (Address address) throws SessionException{
38.   try{
39.      sessionServer.removeAddress(address, sessionID);
40.    }
41.    catch (RemoteException exc)()
42.}
43.  public void commitChanges() throws SessionException{
```

```
44.     try{
45.      sessionID = sessionServer.finalizeContact(sessionID) ;
46.     }
47.     catch (RemoteException exc){}
48.  }
49.}
```

Каждый метод клиента вызывает соответствующий метод удаленного сервера. Интерфейс SessionServer (листинг А.205) определяет четыре метода, доступные клиентам через механизм RMI.

Листинг А.205. SessionServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface SessionServer extends Remote{
4.   public long addContact(Contact contact, long sessionID) throws
   RemoteException, SessionException;
5.   public long addAddress(Address address, long sessionID) throws
   RemoteException, SessionException;
6.   public long removeAddress(Address address, long sessionID) throws
   RemoteException, SessionException;
7.   public long finalizeContact(long sessionID) throws RemoteException,
   SessionException;
8. }
```

Класс SessionServerImpl (листинг А.206) реализует интерфейс SessionServer, обеспечивая тем самым создание сервера RMI. Класс SessionServerImpl делегирует определение бизнес-логики классу SessionServerDelegate (листинг А.207).

Листинг А.206. SessionServerImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. public class SessionServerImpl implements SessionServer{
4.   private static final String SESSION_SERVER_SERVICE_NAME = "sessionServer";
5.   public SessionServerImpl(){
6.      try)
7.       UnicastRemoteObject.exportObject(this) ;
8.       Naming.rebind(SESSION_SERVER_SERVICE_NAME, this);
9.      }
10.     catch (Exception exc){
11.       System.err.println("Error using RMI to register the SessionServerImpl"
   + exc) ;
12.     }
13.  }
14.
15.  public long addContact(Contact contact, long sessionID) throws
   SessionException{
16.     return SessionServerDelegate.addContact(contact, sessionID);
17.  )
18.
19.  public long addAddress(Address address, long sessionID) throws
   SessionException{
20.     return SessionServerDelegate.addAddress(address, sessionID);
21.  }
22.
```

```
23. public long removeAddress(Address address, long sessionID) throws
    SessionException{
24.   return SessionServerDelegate.removeAddress(address, sessionID);
25. }
26.
27. public long finalizeContact(long sessionID) throws SessionException{
28.   return SessionServerDelegate.finalizeContact(sessionID) ;
29. }
30.}
```

Листинг А.207. SessionServerDelegate.java

```
1. import java.util.ArrayList;
2. import java.util.HashMap;
3. public class SessionServerDelegate{
4.   private static final long NO_SESSION_ID = 0;
5.   private static long nextSessionID = 1;
6.   private static ArrayList contacts = new ArrayList ();
7.   private static ArrayList addresses = new ArrayList();
8.    private static HashMap editContacts = new HashMap();
9.
10.  public static long addContact(Contact contact, long sessionID) throws
     SessionException{
11.     if (sessionID <= NO_SESSION_ID){
12.       sessionID = getSessionID();
13.     }
14.     if (contacts.indexOf(contact) != -1){
15.       if (!editContacts.containsValue(contact)){
16.           editContacts.put(new Long(sessionID), contact);
17.       }
18.       else{
19.       throw new SessionException("This contact is currently being edited by
     another user.",
20.           SessionException.CONTACT_BEING_EDITED) ;
21.       }
22.     }
23.     else{
24.       contacts.add(contact);
25.       editContacts.put(new Long(sessionID), contact;
26.     }
27.     return sessionID;
28.  }
29.
30.  public static long addAddress(Address address, long sessionID) throws
     SessionException{
31.     if (sessionID <= NO_SESSION_ID){
32.       throw new SessionException("A valid session ID is required to add an
     address",
33.           SessionException.SESSION_ID_REQUIRED);
34.     >
35.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
36.     if (contact == null){
37.       throw new SessionException("You must select a contact before adding an
     address",
38.           SessionException.CONTACT_SELECT_REQUIRED);
39.     }
40.     if (addresses.indexOf(address) == -1){
41.       addresses.add(address);
42.     }
43.     contact.addAddress(address);
44.     return sessionID;
```

```
45. }
46.
47. public static long removeAddress(Address address, long sessionID) throws
    SessionException{
48.     if (sessionID <= NO_SESSION_ID){
49.       throw new SessionException("A valid session ID is required to remove
    an address",
50.             SessionException.SESSION_ID_REQUIRED);
51.     }
52.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
53.     if (contact == null){
54.       throw new SessionException("You must select a contact before removing
    an address",
55.             SessionException.CONTACT_SELECT_REQUIRED);
56.     }
57.     if (address.indexOf(address) == -1){
58.       throw new SessionException("There is no record of this address",
59.           SessionException.ADDRESS_DOES_NOT_EXIST) ;
60.     }
61.     contact.removeAddress(address);
62.     return sessionID;
63. }
64.
65. public static long finalizeContact(long sessionID) throws SessionException{
66.     if (sessionID <= NO_SESSION_ID){
67.       throw new SessionException("A valid session ID is required to finalize
    a contact",
68.             SessionException.SESSION_ID_REQUIRED);
69.     }
70.     Contact contact = (Contact)editContacts.get(new Long(sessionID));
71.     if (contact == null){
72.       throw new SessionException("You must select and edit a contact before
    committing changes",
73.             SessionException.CONTACT_SELECT_REQUIRED);
74.     }
75.     editContacts.remove(new Long(sessionID));
76.     return NO_SESSION_ID;
77. }
78.
79. private static long getSessionID(){
80.     return nextSessionID++;
81. }
82.
83. public static ArrayList getContacts(){ return contacts; }
84. public static ArrayList getAddresses () { return addresses; }
85.  public static ArrayList getEditContacts(){ return new
    ArrayList (editContacts.values());}
86.}
```

Класс SessionServerDelegate генерирует идентификатор сеанса для клиентов, когда они выполняют свою первую операцию, добавления объекта класса Contact. Для выполнения последующих операций с адресом контакта требуется наличие идентификатора сеанса, поскольку этот идентификатор используется в классе SessionServerDelegate для ассоциации адреса с конкретным объектом класса Contact.

Любые ошибки, которые будут возникать в ходе выполнения примера, обрабатываются с помощью класса SessionException (листинг А.208).

Листинг А.208. SessionException.java

```
1. public class SessionException extends Exception{
2.   public static final int CONTACT_BEING_EDITED = 1;
3.   public static final int SESSION_ID_REQUIRED = 2;
4.   public static final int CONTACT_SELECT_REQUIRED = 3;
5.   public static final int ADDRESS_DOES_NOT_EXIST = 4;
6.   private int errorCode;
7.
8.   public SessionException(String cause, int newErrorCode){
9.      super(cause);
10.     errorCode = newErrorCode;
11.  }
12.  public SessionException(String cause){ super(cause); }
13.
14.  public int getErrorCode(){ return errorCode; }
15.}
```

Прикладные объекты контактов, хранящихся в адресной книге, и соответствующих им адресов, представлены интерфейсами Address (листинг А.209) и Contact (листинг А.211), а также реализующими их классами `AddressImpl` (листинг А.210) и `ContactImpl` (листинг А.212).

Листинг А.209. Address.java

```
1. import java.io.Serializable;
2. public interface Address extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.   public static final String COMMA = ",";
6.   public String getType();
7.   public String getDescription();
8.   public String getStreet();
9.   public String getCity();
10.  public String getState();
11.  public String getZipCode();
12.
13.  public void setType(String newType);
14.  public void setDescription(String newDescription);
15.  public void setStreet(String newStreet);
16.  public void setCity(String newCity);
17.  public void setState (String newState);
18.  public void setZipCode(String newZip);
19.}
```

Листинг А.210. AddressImpl.java

```
1. public class AddressImpl implements Address{
2.   private String type;
3.   private String description;
4.   private String street;
5.   private String city;
6.   private String state;
7.   private String zipCode;
8.
```

```
9.  public AddressImpl(){ }
10. public AddressImpl(String newDescription, String newStreet,
11.    String newCity, String newState, String newZipCode){
12.      description = newDescription;
13.      street = newStreet;
14.    • city = newCity;
15.      state = newState;
16.      zipCode = newZipCode;
17. }
18.
19. public String getType(){ return type; }
20. public String getDescription(){ return description; }
21. public String getStreet(){ return street; }
22. public String getCity(){ return city; }
23. public String getState(){ return state; }
24. public String getZipCode(){ return zipCode; }
25.
26. public void setType(String newType){ type = newType; }
27. public void setDescription(String newDescription){ description =
    newDescription; }
28. public void setStreet (String newStreet) { street = newStreet; }
29. public void setCity(String newCity){ city = newCity; }
30. public void setState(String newState){ state = newState; }
31. public void setZipCode(String newZip){ zipCode = newZip; }
32.
33. public boolean equals(Object o){
34.    if (!(o instanceof AddressImpl)){
35.      return false;
36.    }
37.    else{
38.      AddressImpl address = (AddressImpl)o;
39.      if (street.equals(address.street) &&
40.          city.equals(address.city) &&
41.          state.equals(address.state) SS
42.         zipCode.equals(address.zipCode) ) {
43.          return true;
44.      }
45.      return false;
46.    }
47. }
48.
49. public String toString(){
50.    return street + EOL_STRING + city + COMMA + SPACE +
51.      state + SPACE + zipCode + EOL_STRING;
52. }
53.}
```

Листинг А.211. Contact.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public interface Contact extends Serializable{
4.   public static final String SPACE = " ";
5.   public static final String EOL_STRING =
     System.getProperty("line.separator") ;
6.   public String getFirstName();
7.   public String getLastName();
8.   public String getTitleO;
9.   public String getOrganization();
10.  public ArrayList getAddresses();
11.
```

```
12. public void setFirstName(String newFirstName);
13. public void setLastName(String newLastName);
14. public void setTitle(String newTitle);
15. public void setOrganization(StringnewOrganization) ;
16. public void addAddress(Address address);
17. public void removeAddress(Address address);
18.)
```

Листинг А.212. ContactImpl.java

```
1. import java.util.ArrayList;
2. public class ContactImpl implements Contact{
3.   private String firstName;
4.   private String lastName;
5.   private String title;
6.   private String organization;
7.   private ArrayList addresses = new ArrayList ();
8.
9.   public ContactImpl(){}
10.  public ContactImpl(String newFirstName, String newLastName,
11.    String newTitle, String newOrganization, ArrayList newAddresses){
12.      firstName = newFirstName;
13.      lastName = newLastName;
14.      title = newTitle;
15.      organization = newOrganization;
16.     if (newAddresses != null){ addresses = newAddresses; }
17. }
18.
19. public String getFirstName(){ return firstName; }
20. public String getLastName(){ return lastName; }
21. public String getTitle(){ return title; }
22. public String getOrganization(){ return organization; }
23. public ArrayList getAddresses(){ return addresses; }
24.
25. public void setFirstName(String newFirstName)) firstName = newFirstName; }
26. public void setLastName(String newLastName){ lastName = newLastName; }
27. public void setTitle(String newTitle){ title = newTitle; }
28. public void setOrganization(String newOrganization)) organization =
    newOrganization; }
29. public void addAddress(Address address))
30.    if(!addresses.contains(address)){
31.     addresses.add(address) ;
32.    }
33. }
34. public void removeAddress(Address address))
35.    addresses.remove(address);
36. }
37.
38. public boolean equals(Object o){
39.    if (!(o instanceof ContactImpl)))
40.      return false;
41.    }
42.    else)
43.      ContactImpl contact = (ContactImpl)o;
44.      if (firstName.equals(contact.firstName) &&
45.          lastName.equals(contact.lastName) &&
46.          organization.equals(contact.organization) &&
47.           title.equals(contact.title)){
48.           return true;
49.       )
```

```
50.       return false;
51.     }
52.  }
53.
54.  public String toString(){
55.     return firstName + SPACE + lastName + EOL_STRING + addresses;
56.  }
57.}
```

Демонстрация реализации на практике обмена информацией между клиентами и серверами с помощью объектов сеансов осуществляется классом RunPattern (листинг А.213). В методе main этого класса сначала создается сервер и два клиента, а затем оба клиента используются для редактирования объектов класса Contact путем добавления и удаления объектов класса Address.

Листинг А.213. RunPattern.java

```
1. import java.io.IOException;
2. public class RunPattern{
3.   public static void main(String [] arguments){
4.     System.out.println("Example for the Session pattern");
5.     System.out.println("This demonstration will show how a Session can be used");
6.     System.out.println (" to organize a series of actions between a client and");
7.     System.out.println(" server.");
8.     System.out.println("In this case, clients will use sessions to coordinate");
9.     System.out.println(" edits of Contact addresses.");
10.     System.out.println();
11.
12.     System.out.println("Running the RMI compiler (rmic)");
13.     System.out.println();
14.   try{
15.     Process p1 = Runtime.getRuntime() .exec("rmic SessionServerImpl" ) ;
16.     p1 .waitFor () ;
17.     }
18.     catch (IOException exc) {
19.       System.err.println ("Unable to run rmic utility. Exiting application.");
20.       System.exit (1) ;
21.     }
22.     catch (InterruptedException exc) {
23.       System.err.println("Threading problems encountered while using the
   rmic utility.");
24.     }
25.
26.     System.out.println("Starting the rmiregistry");
27.     System.out.println ();
28.     Process rmiProcess = null;
29.   try{
30.       rmiProcess = Runtime.getRuntime().exec("rmiregistry");
31.       Thread.sleep(15000);
32.     }
33.     catch (IOException exc) {
34.        System.err.println("Unable to start the rmiregistry. Exiting application.");
35.       System.exit(1);
36.     )
37.      catch (InterruptedException exc) {
38.       System.err.println("Threading problems encountered when starting the
   rmiregistry.");
39.     }
40.
```

```
41.   System.out.println("Creating the SessionServer and two SessionClient
    objects");
42.    System.out.println();
43.    SessionServer serverObject = new SessionServerImpl();
44.    SessionClient clientOne = new SessionClient();
45.    SessionClient clientTwo = new SessionClient();
46.
47.    System.out.println("Creating sample Contacts and Addresses");
48.    System.out.println();
49.   Contact firstContact = new ContactImpl("First", "Contact", "primo",
    "OOI",null);
50.   Contact secondContact = new ContactImpl("Second", "Contact", "secondo",
    "OOI",null);
51.   Address workAddress = new AddressImpl("Work address", "5440 Division",
    "Fargo", "ND", "54321");
52.   Address homeAddress = new AddressImpl("Home address", "40 Planar Way",
    "Paris", "TX", "84301");
53.
54.   System.out.println("Adding a contact. Both clients will attempt to edit");
55.   System.out.println(" the same contact at first, which will result in a");
56.    System.out.println(" SessionException.");
57.    try{
58.     clientOne.addContact(firstContact) ;
59.     clientTwo.addContact(firstContact) ;
60.    )
61.    catch (SessionException exc){
62.     System.err.println("Exception encountered:");
63.     System.err.println(exc) ;
64.    }
65.    try{
66.    System.out.println("Adding a different contact to the second client");
67.   clientTwo.addContact(secondContact) ;
68.    System.out.println ("Adding addresses to the first and second clients");
69.     clientTwo.addAddress(workAddress);
70.     clientOne.addAddress(homeAddress);
71.     clientTwo.addAddress(workAddress);
72.     clientTwo.addAddress(homeAddress);
73.    System.out.println("Removing address from a client");
74.     clientTwo.removeAddress(homeAddress);
75.    System.out.println("Finalizing the edits to the contacts");
76.    clientOne.commitChanges() ;
77.     clientTwo.commitChanges();
78.    System.out.println("Changes finalized");
79.    clientTwo.addContact(firstContact) ;
80.    }
81.   catch (SessionException exc) {
82.     System.err.println("Exception encountered:");
83.     System.err.println(exc) ;
84.    }
85.   System.out.println("The following lines will show the state");
86.   System.out.println(" of the server-side delegate, which in this");
87.   System.out.println(" example represents a persistent data store.");
88.    System.out.println();
89.   System.out.println("Contact list:");
90.    System.out.println(SessionServerDelegate.getContacts()) ;
91.    System.out.println("Address list:");
92.    System.out.println(SessionServerDelegate.getAddresses());
93.    System.out.println("Edit contacts:");
94. System.out.println(SessionServerDelegate.getEditContacts() ) ;
95. )
96.}
```

Worker Thread

Допустим в типичном приложении нужно выполнить какую-то определенную работу. Иногда даже не важно, какая именно эта работа, главное, чтобы она выполнялась. Это можно сравнить с уборкой дома. Если кто-то занялся в эту субботу уборкой, то уже не важно, что он делает в данный конкретный момент времени, лишь бы он это делал.

В данном примере для хранения задач используется очередь, представленная интерфейсом Queue (листинг А.214). Интерфейс Queue определяет два базовых метода: put и take. Эти методы используются для добавления в очередь и удаления из нее задач, представленных интерфейсом RunnableTask (листинг А.215).

Листинг А.214. Queue.java

```
1. public interface Queue{
2.  void put(RunnableTask r) ;
3.   RunnableTask take();
4. )
```

Листинг А.215. RunnableTask.java

```
1. public interface RunnableTask{
2.   public void execute();
3. }
```

Класс ConcreteQueue (листинг А.216) реализует интерфейс Queue и создает обработчик потока, который выполняет объекты, реализующие интерфейс RunnableTask. Класс Worker, который является внутренним классом класса ConcreteQueue, имеет метод run, предназначенный для периодической проверки очереди на наличие в ней готовых к выполнению задач. Когда такая задача появляется, обработчик потока извлекает ее из очереди и запускает ее метод execute.

Листинг А.216. ConcreteQueue.java

```
1. import java.util.Vector;
2. public class ConcreteQueue implements queue{
3.  private Vector tasks = new Vector ();
4.  private boolean waiting;
5.  private boolean shutdown;
6.
7.  public void setShutdown(boolean isShutdown){ shutdown = isShutdown; }
8.
9.  public ConcreteQueue (){
10.    tasks = new Vector)):
11.    waiting = false;
12.   new Thread (new Worker ()). start ();
13. }
14.
15. public void put(RunnableTask r){
16.    tasks.add(r);
17.    if (waiting){
18.      synchronized (this){
```

```
19.          notifyAll () ;
20.       }
21.    }
22. }
23.
24. public RunnableTask take(){
25.  if (tasks. isEmpty () ) {
26.      synchronized (this){
27.          waiting = true;
28.          try{
29.              wait();
30.          } catch (InterruptedException ie){
31.              waiting = false;
32.           }
33.       }
34.    }
35.    return (RunnableTask)tasks.remove(0) ;
36. }
37.
38. private class Worker implements Runnable{
39.    public void run(){
40.      while (!shutdown){
41.          RunnableTask r = take();
42.         r .execute () ;
43.       }
44.    }
45. )
46.}
```

В данном примере используются два класса, реализующие интерфейс RunnableTask, — AddressRetriever (листинг А.217) и ContactRetriever (листинг А.218). Классы очень похожи друг на друга, оба используют технологию RMI для запроса бизнес-объекта у сервера. Как видно из их названий, каждый класс получает от сервера бизнес-объекты определенного типа, делая доступными клиентам объекты класса Address и Contact, соответственно.

Листинг А.217. AddressRetriever.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. public class AddressRetriever implements RunnableTaskf
4.   private Address address;
5.   private long addressID;
6.   private String url;
7.
8.   public AddressRetriever(long newAddressID, String newUrl){
9.      addressID = newAddressID;
10.     url = newUrl;
11.  )
12.
13.  public void execute(){
14.     try{
15.       ServerDataStore dataStore =  (ServerDataStore)Naming.lookup(url),
16.       address = dataStore.retrieveAddress(addressID);
17.     }
18.    catch (Exception exc) (
19.     }
20. }
21.
```

```
22.  public Address getAddress(){ return address; }
23.  public boolean isAddressAvailable (){ return (address == null) ? false : true; }
24.}
```

Листинг А.218. ContactRetriever.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. public class ContactRetriever implements RunnableTask{
4.  private Contact contact;
5.  private long contactID;
6.  private String url;
7.
8.  public ContactRetriever(long newContactID, String newUrl){
9.     contactID = newContactID;
10.    url = newUrl;
11.  }
12.
13.  public void execute(){
14.   try(
15.      ServerDataStore dataStore = (ServerDataStore)Naming.lookup(url);
16.     contact = dataStore.retrieveContact(contactID) ;
17.   }
18.   catch (Exception exc) {
19.    }
20. >
21.
22.  public Contact getContact(){ return contact; }
23.  public boolean isContactAvailable(){ return (contact == null) ? false : true; }
24.}
```

В данном примере сервер RMI представлен интерфейсом ServerDataStore (листинг А.219) и реализующим его классом ServerDataStoreImpl (листинг А.220).

Листинг А.219. ServerDataStore.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface ServerDataStore extends Remote{
4.  public Address retrieveAddress(long addressID) throws RemoteException;
5.  public Contact retrieveContact(long contactID) throws RemoteException;
6. }
```

Листинг А.220. ServerDataStoreImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. public class ServerDataStoreImpl implements ServerDataStore{
4.  private static final String WORKER_SERVER_SERVICE_NAME =
   "workerThreadServer";
5.
6.  public ServerDataStoreImpl(){
7.    try {
8.       UnicastRemoteObject.exportObject(this);
9.      Naming.rebind(WORKER_SERVER_SERVICE_NAME, this);
```

```
ю.  >
11.    catch (Exception exc){
12.      System.err.println("Error using RMI to register the
   ServerDataStoreImpl" + exc);
13.    }
14. }
15.
16. public Address retrieveAddress(long addressID){
17.    if (addressID == 5280L){
18.      return new AddressImpl("Fine Dining", "416 Chartres St.", "New
   Orleans", "LA", "51720");
19.    )
20.    else if (addressID == 2010L){
21.      return new AddressImpl("Mystic Yacht Club", "19 Imaginary Lane",
   "Mystic", "CT", "46802");
22.    }
23.    else{
24.      return new AddressImpl();
25.    }
26. }
27. public Contact retrieveContact(long contactID){
28.    if (contactID == 5280L){
29.     return new ContactImpl("Dwayne", "Dibley", "Accountant", "Virtucon");
30.    }
31.    else{
32.      return new ContactImpl();
33.    )
34. }
35.}
```

Прикладные объекты контактов, хранящихся в адресной книге, и соответствующих им адресов, представлены интерфейсами Address (листинг А.221) и Contact (листинг А.223), а также реализующими их классами Addresslmpl (листинг А.222) и Contactlmpl (листинг А.224).

Листинг А.221. Address.java

```
1. import java.io.Serializable;
2. public interface Address extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.   public static final String COMMA = ",";
6.   public String getType();
7.   public String getDescription ();
8.   public String getStreet();
9.   public String getCity();
10.  public String getState();
11.  public String getZipCode();
12.
13.  public void setType(String newType);
14.  public void setDescription(String newDescription) ;
15.  public void setStreet(String newStreet);
16.  public void setCity(String newCity);
17.  public void setState(String newState);
18.  public void setZipCode(String newZip);
19.}
```

Листинг A.222. AddressImpl.java

```
1. public class AddressImpl implements Address{
2.   private String type;
3.   private String description;
4.   private String street;
5.   private String city;
6.   private String state;
7.   private String zipCode;
8.
9.   public AddressImpl(){ }
10.  public AddressImpl(String newDescription, String newStreet,
11.     String newCity, String newState, String newZipCode){
12.       description = newDescription;
13.       street = newStreet;
14.       city = newCity;
15.       state = newState;
16.       zipCode = newZipCode;
17. }
18.
19.  public String getType(){ return type; }
20.  public String getDescription(){ return description; }
21.  public String getStreet(){ return street; }
22.  public String getCity(){ return city; }
23.  public String getState(){ return state; }
24.  public String getZipCode(){ return zipCode; }
25.
26.  public void setType(String newType){ type = newType; }
27.  public void setDescription(String newDescription){ description =
   newDescription; }
28.  public void setStreet(String newStreet){ street = newStreet; }
29.  public void setCity(String newCity){ city = newCity; }
30.  public void setState(String newState){ state = newState; }
31.  public void setZipCode(String newZip){ zipCode = newZip; }
32.
33.  public String toString(){
34.     return street + EOL_STRING + city + COMMA + SPACE +
35.       state + SPACE + zipCode + EOL_STRING;
36. }
37.}
```

Листинг A.223. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.   public static final String EOL_STRING =
   System.getProperty("line.separator");
4.   public static final String SPACE = " ";
5.    public String getFirstName();
6.    public String getLastName();
7.    public String getTitle();
8.    public String getOrganization ();
9.
10.   public void setFirstName(String newFirstName);
11.   public void setLastName(String newLastName);
12.   public void setTitle(String newTitle);
13.   public void setOrganization(String newOrganization);
14.}
```

Листинг А.224. ContactImpl.java

```
1. public class Contactlmpl implements Contact{
2.   private String firstName;
3.   private String lastName;
4.   private String title;
5.   private String organization;
6.
7.   public ContactImpl(){}
8.   public ContactImpl(String newFirstName, String newLastName,
9.      String newTitle, String newOrganization){
10.       firstName = newFirstName;
11.       lastName = newLastName;
12.       title = newTitle;
13.       organization = newOrganization;
14.  }
15.
16.  public String getFirstName(){ return firstName; )
17.  public String getLastName(){ return lastName; }
18.  public String getTitle(){ return title; )
19.  public String getOrganization(){ return organization; }
20.
21.  public void setFirstName(String newFirstName){ firstName = newFirstName; }
22.  public void setLastName(String newLastName){ lastName = newLastName; }
23.  public void setTitle(String newTitle){ title = newTitle; }
24.  public void setOrganization(String newOrganization){ organization =
   newOrganization; }
25.
26.  public String toString(){
27.     return firstName + SPACE + lastName + EOL_STRING;
28.  }
29.}
```

Класс RunPattern (листинг А.225) создает экземпляр класса ConcreteQueue, а затем использует его для получения одного экземпляра класса Contact и двух экземпляров класса Address. Обработчик потока, встроенный в очередь, обрабатывает эти запросы по мере их поступления в очередь. Экземпляр класса ConcreteQueue и связанный с ним обработчик потока можно использовать в течение всего сеанса работы приложения, возлагая на них любые фоновые задачи, которые выполняются по запросам клиентов. Для того чтобы решить подобную задачу, клиент должен обеспечить, чтобы соответствующий объект реализовывал интерфейс RunnableTask, и поставить задачу в очередь.

Листинг А.225. RunPattern.java

```
1. import java.io.IOException;
2. public class RunPattern(
3.   private static final String WORKER_SERVER_URL =
   "//localhost/workerThreadServer";
4.   public static void main(String [] arguments){
5.      System.out.println("Example for the WorkerThread pattern");
6.     System.out.println("In this example, a ConcreteQueue object which uses a");
7.      System.out.println(" worker thread, will retrieve a number of objects
   from");
8.      System.out.println(" the server.");
9.      System.out.println() ;
10.
```

```
11.    System.out.println("Running the RMI compiler (rmic)");
12.    System.out.println() ;
13.     try{
14.      Process p1 = Runtime.getRuntime().exec("rmic ServerDataStoreImpl");
15.       p1.waitFor();
16.     }
17.    catch (IOException exc) {
18.     System.err.println("Unable to run rmic utility. Exiting application.");
19.     System.exit (1) ;
20.     }
21.    catch (InterruptedException exc){
22.      System.err.println("Threading problems encountered while using the
   rmic utility.");
23.     }
24.
25.    System.out.println("Starting the rmiregistry");
26.     System.out.println();
27.    Process rmiProcess = null;
28.     try{
29.       rmiProcess = Runtime.getRuntime().exec("rmiregistry");
30.       Thread.sleep(15000);
31.     }
32.     catch (IOException exc){
33.      System.err.println("Unable to start the rmiregistry. Exiting
   application.") ;
34.       System.exit(1);
35.     )
36.     catch (InterruptedException exc){
37.      System.err.println("Threading problems encountered when starting the
   rmiregistry.") ;
38.     }
39.
40.    System.out.println("Creating the queue, which will be managed by the
   worker thread");
41.    System.out.println() ;
42.     ConcreteQueue workQueue = new ConcreteQueue();
43.
44.     System.out.println("Creating the RMI server object,
   ServerDataStoreImpl");
45.     System.out.println();
46.     ServerDataStore server = new ServerDataStoreImpl();
47.
48.     System.out.println("Creating AddressRetrievers and ContactRetreivers.");
49.    System.out.println(" These will placed in the queue, as tasks to be");
50.     System.out.println(" performed by the worker thread.");
51.     System.out.println();
52.     AddressRetriever firstAddr = new AddressRetriever(5280L, WORKER_SERVER_URL);
53.     AddressRetriever secondAddr = new AddressRetriever(2010L,
   WORKER_SERVER_URL);
54.     ContactRetriever firstContact = new ContactRetriever(5280L,
   WORKER_SERVER_URL);
55.
56.    workQueue.put(firstAddr) ;
57  .  workQueue.put(firstContact);
58.     workQueue.put(secondAddr);
59.
60.     while (!secondAddr.isAddressAvailable()){
61.       try{
62.           Thread.sleep(1000) ;
63.       }
64.       catch (InterruptedException exc){}
65.     }
66.
```

```
67.   System.out.println("WorkerThread completed the processing of its Tasks");
68.   System.out.println("Printing out the retrieved objects now:");
69.    System.out.println();
70.  System.out.println(firstAddr.getAddress() ) ;
71.  System.out.println(firstContact.getContact() ) ;
72.   System.out.println(secondAddr.getAddress()) ;
73.
74.  }
75.
76.}
```

Callback

В РШ-приложении имеется один элемент, размер которого может сильно изменяться, — это проект. Действительно, проект может состоять из нескольких задач, а может содержать сотни и даже тысячи отдельных этапов. В данном примере показано, как с помощью шаблона Callback можно организовать получение проекта, хранящегося на сервере.

Представленный в листинге А.226 интерфейс `CallbackServer` содержит определение лишь одного метода сервера `getProject`. Обратите внимание на то, что этот метод требует, помимо идентификатора проекта, информацию для обратного вызова: имя клиентской машины и имя клиентского объекта RMI. Данный интерфейс реализуется классом `CallbackServerImpl` (листинг А.227).

Листинг А.226. CallbackServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface CallbackServer extends Remote{
4.  public void getProject(String projectID, String callbackMachine,
5.     String callbackObjectName) throws RemoteException;
6. }
```

Листинг А.227. CallbackServerImpl.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. public class CallbackServerImpl implements CallbackServer{
4.  private static final String CALLBACK_SERVER_SERVICE_NAME -
   "CallbackServer";
5.  public CallbackServerImpl(){
6.     try {
7.       UnicastRemoteObject.exportObject(this);
8.      Naming.rebind(CALLBACK_SERVER_SERVICE_NAME, this);
9.     }
10.    catch (Exception exc) {
11.     System.err.println("Error using RMI to register the
   CallbackServerImpl" + exc) ;
12.    }
13. }
14.
15. public void getProject(String projectID, String callbackMachine,
16.    String callbackObjectName){
```

```
17.     new CallbackServerWorkThread(projectID, callbackMachine,
    callbackObjectName);
18. }
19.
20. }
```

В реализации метода getProject класс CallbackServerImpl делегирует задачу получения проекта рабочему объекту CallbackServerDelegate (листинг А.228). Этот объект выполняется в отдельном потоке и отвечает за извлечение проекта и отправку его клиенту.

Листинг А.228. CallbackServerDelegate.java

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. public class CallbackServerDelegate implements Runnable{
6.   private Thread processingThread;
7.   private String projectID;
8.   private String callbackMachine;
9.   private String callbackObjectName;
10.
11.  public CallbackServerDelegate(String newProjectID, String newCallbackMachine,
12.  String newCallbackObjectName){
13.     projectID = newProjectID;
14.     callbackMachine = newCallbackMachine;
15.     callbackObjectName = newCallbackObjectName;
16.     processingThread = new Thread(this);
17.    processingThread.start() ;
18.  }
19.
20.  public void run(){
21.     Project result = getProject ();
22.     sendProjectToClient(result);
23.  >
24.
25.  private Project getProject (){
26.     return new Project(projectID, "Test project");
27.  }
28.
29.  private void sendProjectToClient(Project project){
30.     try{
31.       String url = "//" + callbackMachine + "/" + callbackObjectName;
32.       Object remoteClient = Naming.lookup(url);
33.       if (remoteClient instanceof CallbackClient){
34.            ((CallbackClient)remoteClient).receiveProject(project);
35.       }
36.     )
37.     catch (RemoteException exc){}
38.     catch (NotBoundException exc){ }
39.     catch (MalformedURLException exc){}
40.  }
41.}
```

В методе run класса CallbackServerDelegate осуществляется извлечение проекта с помощью вызова метода getProject с последующей отправкой его клиенту путем вызова метода sendProjectToClient. Последний метод осуществляет обратный вы-

зов клиента— объект CallbackServerDelegate вызывает объект RMI типа CallbackClient, находящийся на клиентской машине. Интерфейс CallbackClient (листинг А.229) также определяет один метод RMI receivePrcject.

Листинг А.229. CallbackClient.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface CallbackClient extends Remote{
4.   public void receiveProject(Project project)  throws RemoteException;
5. }
```

Класс CallbackClientImpl (листинг А.230), реализующий интерфейс CallbackClient, является одновременно и клиентом, и сервером. Его метод requestProject обращается к объекту класса CallbackServer и вызывает удаленный метод getProject. Класс также определяет удаленный метод receiveProject, который вызывается обработчиком потока сервера, когда проект готов для предоставления его клиенту. Класс CallbackClientImpl имеет булеву переменную projectAvailable, которая позволяет клиентской программе определить, готов ли проект для отображения.

Листинг А.230. CallbackClientImpl.java

```
1. import java.net.InetAddress;
2. import java.net.MalformedURLException;
3. import java.net.UnknownHostException;
4. import java.rmi.Naming;
5. import java.rmi.server.UnicastRemoteObject;
6. import java.rmi.NotBoundException;
7. import java.rmi.RemoteException;
8. public class CallbackClientImpl implements CallbackClient{
9.   private static final String CALLBACK_CLIENT_SERVISE_NAME =
   "CallbackClient";
10.  private static final String CALLBACK_SERVER_SERVICE_NAME =
   "callbackServer";
11.  private static final String CALLBACK_SERVER_MACHINE_NAME = "localhost";
12.
13.  private Project requestedProject;
14.  private boolean projectAvailable;
15.
16.  public CallbackClientImpl(){
17.    try {
18.       UnicastRemoteObject(this);
19.      Naming.rebind(CALLBACK_CLIENT_SERVISE_NAME, this);
20.    }
21.     catch (Exception exc){
22.       System.err.println("Error using RMI to register the
   CallbackClientImpl" + exc);
23.    }
24.  }
25.
26.  public void receiveProject (Project project){
27.     requestedProject = project;
28.     projectAvailable = true;
29.  }
30.
```

```
31. public void requestProject(String projectName){
32.   try)
33.     String url = "//" + CALLBACK_SERVER_MACHINE_NAME +"/" +
   CALLBACK_SERVER_SERVICE_NAME;
34.     Object remoteServer = Naming.lookup(url);
35.     if (remoteServer instanceof CallbackServer){
36.           ((CallbackServer)remoteServer).getProject(projectName,
37.               InetAddress.getLocalHost().getHostName(),
38.               CALLBACK_CLIENT_SERVICE_NAME);
39.      }
40.     projectAvailable = false;
41.    }
42.    catch (RemoteException exc) {}
43.   catch (NotBoundException exc){ }
44.    catch (MalformedURLException exc) {}
45.    catch (UnknownHostException exc) {}
46. }
47.
48.  public Project getProject(){ return requestedProject; }
49.  public boolean isProjectAvailable(){ return projectAvailable; }
50.}
```

Основные операции происходят в следующей последовательности. Когда клиент запрашивает проект, объект класса `CallbackClientImpl` вызывает метод `getProject` объекта класса `CallbackServerImpl`. Последний создает для извлечения проекта объект `CallbackServerWorkThread`. После того как поток `CallbackServerWorkThread` завершит свою работу, он вызывает метод клиента `receiveProject`, отправляя экземпляр класса `Project` запрашивавшему его объекту `CallbackClientImpl`.

В данном примере для представления ресурсов проекта используются интерфейс `ProjectItem` (листинг А.232), а также классы `Project` (листинг А.231) и `Task` (листинг А.233).

Листинг А.231. Project.java

```
1. import java.util.ArrayList;
2. public class Project implements ProjectItem{
3.   private String name;
4.   private String description;
5.   private ArrayList projectItems = new ArrayList ();
6.
7.   public Project (){ )
8.   public Project(String newName, String newDescription){
9.     name = newName;
10.    description = newDescription;
11. }
12.
13.  public String getName(){ return name; )
14.  public String getDescription(){ return description; }
15.  public ArrayList getProjectItems(){ return projectItems; }
16.
17.  public void setName(String newName){ name = newName; }
18.  public void setDescription(String newDescription){ description =
   newDescription; }
19.
20.  public void addProjectItem(ProjectItem element){
21.    if (!projectItems.contains(element)){
22.      projectItems.add(element);
23.    }
```

```
24. }
25.
26. public void removeProjectItem(ProjectItem element){
27.   projectItems.remove(element) ;
28. }
29.
30. public String toString(){ return name + ", " + description; }
31.}
```

Листинг А.232. ProjectItem.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. public interface ProjectItem extends Serializable{
4.   public ArrayList getProjectItems();
5. )
```

Листинг А.233. Task.java

```
1. import java.util.ArrayList;
2. public class Task implements ProjectItem{
3.   private String name;
4.   private ArrayList projectItems = new ArrayList ();
5.   private double timeRequired;
6.
7.   public Task() { }
8.   public Task(String newName, double newTimeRequired){
9.     name = newName;
10.    timeRequired = newTimeRequired;
11.  }
12.
13.  public String getName(){ return name; }
14.  public ArrayList getProjectItems(){ return projectItems; }
15.  public double getTimeRequired(){ return timeRequired; }
16.
17.  public void setName(String newName){ name = newName; }
18.  public void setTimeRequired(double newTimeRequired){ timeRequired =
     newTimeRequired; }
19.
20.  public void addProjectItem(ProjectItem element){
21.    if (!projectItems.contains(element)){
22.      projectItems.add(element);
23.    }
24.  }
25.
26.  public void removeProjectItem(ProjectItem element){
27.    projectItems.remove(element);
28.  }
29.}
```

Демонстрация реализации шаблона выполняется классом `RunPattern` (листинг А.234), который создает объекты клиента и сервера RMI. В данном примере главный поток программы использует экземпляр класса CallbackClientImpl для обращения с запросом к серверу о получении проекта, а затем переходит к выполнению цикла до тех пор, пока не будет получен объект проекта.

Листинг А.234. RunPattern.java

```
1. import java.io.IOException;
2. public class RunPattern{
3.  public static void main(String [] arguments){
4.    System.out.println("Example for the Callback pattern");
5.    System.out.println("This code will run two RMI objects to demonstrate");
6.    System.out.println(" callback capability. One will be CallbackClientImpl,");
7.    System.out.println(" which will request a project from the other remote");
8.    System.out.println(" object, CallbackServerImpl.");
9.    System.out.println("To demonstrate how the Callback pattern allows the");
10.   System.out.println(" client to perform independent processing, the main");
11.   System.out.println(" program thread will go into a wait loop until the");
12.   System.out.println(" server sends the object to its client.");
13.    System.out.println();
14.
15.   System.out.println("Running the RMI compiler (rmic)");
16.    System.out.println();
17.    try{
18.     Process p1 = Runtime.getRuntime().exec("rmic CallbackServerImpl");
19.     Process p2 - Runtime.getRuntime().exec("rmic CallbackClientImpl");
20.    p1.waitFor () ;
21.     p2.waitFor() ;
22.    }
23.   catch (lOException exc) {
24.    System.err.println("Unable to run rmic utility. Exiting application.");
25.    System.exit (1) ;
26.    )
27.   catch (InterruptedException exc) (
28.     System.err.println("Threading problems encountered while using the
   rmicutility.");
29.    }
30.
31.   System.out.println("Starting the rmiregistry");
32.    System.out.println();
33.   Process rmiProcess = null;
34.    try{
35.      rmiProcess = Runtime.getRuntime().exec("rmiregistry");
36.      Thread.sleep(15000);
37.    )
38.   catch (lOException exc) {
39.     System.err.println("Unable to start the rmiregistry. Exiting
   application.");
40.      System.exit (1);
41.    )
42.    catch (InterruptedException exc){
43.     System.err.println("Threading problems encountered when starting the
   rmiregistry.") ;
44.    }
45.
46.   System.out.println("Creating the client and server objects");
47.    System.out.println();
48.   CallbackServerImpl callbackServer = new CallbackServerImpl();
49.   CallbackClientImpl callbackClient - new CallbackClientImpl();
50.
51.    System.out.println("CallbackClientImpl requesting a project");
52.    callbackClient.requestProject("New Java Project");
53.
54.    try{
55.      while (!callbackClient.isProjectAvailable()){
56.          System.out.println("Project not available yet; sleeping for 2
   seconds");
```

```
57.          Thread.sleep(2000) ;
58.       }
59.     }
60.     catch (InterruptedException exc){}
61.    System.out.println("Project retrieved: " + callbackClient.getProject());
62. }
63.)
```

Successive Update

В данном примере показана простая реализация технологии запроса обновления клиентом в PIM-приложении. Клиенты используют сервер для централизованного хранения информации о задачах, над которыми они работают. Каждый клиент следит за актуальностью своих данных, периодически запрашивая у сервера обновление.

В листинге А.235 приведен пример программного кода, представленного классом `PullClient`, который получает для клиента сведения о задаче. Этот класс отвечает за поиск сервера RMI, через который он может постоянно получать информацию о задаче.

Листинг А.235. `PullClient.java`

```
1. import java.net.MalformedURLException;
2. import java.rmi.Naming;
3. import java.rmi.NotBoundException;
4. import java.rmi.RemoteException;
5. import java.util.Date;
6. public class PullClient{
7.   private static final String UPDATE_SERVER_SERVISE_NAME = "updateServer";
8.   private static final String UPDATE_SERVER_MACHINE_NAME = "localhost";
9.   private ClientPullServer updateServer;
10.  private ClientPullRequester requester;
11.  private Task updatedTask;
12.  private String clientName;
13.
14.  public PullClient(String newClientName){
15.     clientName = newClientName;
16.    try{
17.      String url = "//" + UPDATA_SERVER_MACHINE_NAME + "/" +
   UPDATE_SERVER_SERVICE_NAME;
18.       updateServer = (ClientPullServer)Naming.lookup(url);
19.     }
20.     catch (RemoteException exc){}
21.    catch (NotBoundException exc) { }
22.     catch (MalformedURLException exc ({}
23.     catch (ClassCastException exc){}
24. }
25.
26.  public void requestTask(String taskID){
27.    requester = new ClientPullRequester(this, updateServer, taskID);
28. }
29.
30.  public void updateTask(Task task){
31.    requester.updateTask(task) ;
32. }
33.
34.  public Task getUpdatedTask(){
35.     return updatedTask;
36. }
```

```
37.
38.  public void setUpdatedTask(Task task){
39.     updatedTask = task;
40.     System.out.println(clientName + ": received updated task: " + task);
41.  }
42.
43.  public String toString(){
44.     return clientName;
45.  }
46.}
```

Когда клиент желает получить обновление по текущей задаче, он вызывает метод `requestTask` экземпляра класса `PullClient`. Объект этого класса создает обработчик потока (см. раздел "Worker Thread" на стр. 507), класс которого представлен классом `ClientPullRequester` (листинг A.236). Соответствующий объект потока выполняется на стороне клиента и регулярно обращается к серверу с запросами на обновление информации о задаче.

Листинг A.236. ClientPullRequester.java

```
1.  import java.rmi.RemoteException;
2.  public class ClientPullRequester implements Runnable{
3.    private static final int DEFAULT_POLLING_INTERVAL = 10000;
4.    private Thread processingThread;
5.    private PullClient parent;
6.    private ClientPullServer updateServer;
7.    private String taskID;
8.    private boolean shutdown;
9.    private Task currentTask = new TaskImpl();
10.   private int pollingInterval = DEFAULT_POLLING_INTERVAL;
11.
12.   public ClientPullRequester(PullClient newParent, ClientPullServer
    newUpdateServer,
13.    String newTaskID){
14.      parent = newParent;
15.      taskID = newTaskID;
16.      updateServer = newUpdateServer;
17.      processingThread = new Thread(this);
18.      processingThread.start();
19.   }
20.
21.   public void run() {
22.      while (!isShutdown()){
23.        try{
24.        currentTask = updateServer.getTask(taskID,
    currentTask.getLastEditDate());
25.            parent.setUpdateTask(currentTask) ;
26.        }
27.        catch (RemoteException exc){ }
28.        catch (UpdateException exc){
29.           System.out.println(" " + parent + ": " + exc.getMessage());
30.        }
31.        try{
32.             Thread.sleep(pollingInterval);
33.        }
34.        catch (InterruptedException exc){}
35.      }
36.   }
37.
```

```
38. public void updateTask(Task changedTask){
39.    try(
40.     updateServer.updateTask(taskID, changedTask);
41.    }
42.    catch (RemoteException exc) ()
43.    catch (UpdateException exc){
44.    System.out.println("  " + parent + ": " + exc.getMessage());
45.    }
46. }
47.
48. public int getPollingInterval(){ return pollingInterval; }
49. public boolean isShutdown(){ return shutdown; }
50.
51. public void setPollingInterval(int newPollingInterval){ pollingInterval =
    newPollingInterval; }
52. public void setShutdown(boolean isShutdown){ shutdown = isShutdown; }
53.}
```

Поведение сервера RMI определяется интерфейсом `ClientPullServer` (листинг А.237), а управление им осуществляется с помощью класса `ClientPullServerImpl` (листинг А.238). Два метода `getTask` и `updateTask` позволяют клиентам взаимодействовать с сервером.

Листинг А.237. ClientPullServer.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. import java.util.Date;
4. public interface ClientPullServer extends Remote)
5.  public Task getTask(String taskID, Date lastUpdate) throws RemoteException,
    UpdateException;
6.  public void updateTask(String taskID, Task updateTask) throws
    RemoteException, UpdateException;
7. }
```

Листинг А.238. ClientPullServerImpl.java

```
1. import java.util.Date;
2. import java.rmi.Naming;
3. import java.rmi.server.UnicastRemoteObject;
4. public class ClientPullServerImpl implements ClientPullServer)
5.  private static final String UPDATE_SERVER_SERVICE_NAME = "updateServer";
6.  public ClientPullServerImpl(){
7.     try)
8.       UnicastRemoteObject.exportObject(this);
9.      Naming.rebind(UPDATE_SERVER_SERVICE_NAME, this);
10.    }
11.    catch (exception exc)[
12.     System.err.println("Error using RMI to register the
    ClientPullServerImpl " + exc;
13.    }
14. >
15.
16. public Task getTask(String taskID, date lastUpdate) throws UpdateException)
17.    return UpdateServerDelegate.getTask(taskID, lastUpdate);
18. }
19.
```

```
20. public void updateTask(String taskID, Task updatedTask) throws
    UpdateException{
21.    UpdateServerDelegate.updateTask(taskID, updatedTask);
22. }
23.}
```

Поведение сервера, представленного классом `ClientPullServerImpl`, определяет класс UpdateServerDelegate (листинг А.239). Именно он извлекает объекты класса Task, а также обеспечивает передачу обновленных копий этих объектов клиентам, выполняя сравнение последнего обновления с текущим обновлением по значению атрибута Date.

Листинг А.239. UpdateServerDelegate.java

```
1. import java.util.Date;
2. import java.util.HashMap;
3. public class UpdateServerDelegate{
4.  private static HashMap tasks = new HashMap ();
5.
6.  public static Task getTask(String taskID, Date lastUpdate) throws
    UpdateException{
7.      if (tasks.containsKey(taskID)){
8.       Task storedTask = (Task) tasks.get(taskID) ;
9.      if (storedTask.getLastEditDate() .after(lastUpdate) ) {
10.           return storedTask;
11.       }
12.       else{
13.          throw new UpdateException("Task" + taskID + "does not need to be
    updated", UpdateException.TASK_UNCHANGED);
14.       }
15.     }
16.     else{
17.      return loadNewTask(taskID) ;
18.     }
19. }
20.
21. public static void updateTask(String taskID, Task task) throws
    UpdateException{
22.     if (tasks.containsKey(taskID)){
23.       if
    (task.getLastEditDate().equals(((Task)tasks.get(taskID)).getLastEditDate()))
    (
24.         ( (TaskImpl)task) . setLastEditDate (new DateO);
25.          tasks.put(taskID, task);
26.       }
27.       else{
28.          throw new UpdateException("Task " + taskID + " data must be
    refreshed before editing", UpdateException.TASK_OUT_OF_DATE);
29.       }
30.     }
31. }
32.
33. private static Task loadNewTask(String taskID){
34.    Task newTask = new TaskImpl(taskID, "", new DateO, null);
35.    tasks.put(taskID, newTask);
36.     return newTask;
37. }
38.}
```

Интерфейс Task (листинг А.240) и класс TaskImpl (листинг А.241) представляют прикладной объект, который в данном примере моделирует определенный этап проекта. Для идентификации проблем, возникающих в ходе периодического выполнения клиентом запроса обновлений, используется класс UpdateException (листинг А.242).

Листинг А.240. Task.java

```
1. import java.util.Date;
2. import java.io.Serializable;
3. import java.util.ArrayList;
4. public interface Task extends Serializable{
5.   public String getTaskIDO;
6.   public Date getLastEditDate();
7.   public String getTaskName ();
8.   public String getTaskDetails();
9.   public ArrayList getSubTasks() ;
10.
11.  public void setTaskName(String newName);
12.  public void setTaskDetails(String newDetails);
13.  public void addSubTask(Task task);
14.  public void removeSubTask(Task task) ;
15.}
```

Листинг А.241. TaskImpl.java

```
1. import java.util.Date;
2. import java.io.Serializable;
3. import java.util.ArrayList;
4. public class TaskImpl implements Task)
5.   private String taskID;
6.   private Date lastEditDate;
7.   private String taskName;
8.   private String taskDetails;
9.   private ArrayList subTasks = new ArrayList ();
10.
11.  public TaskImpl(){
12.     lastEditDate = new Date () ;
13.     taskName = "";
14.     taskDetails = "";
15.  }
16.  public TaskImpl(String newTaskName, String newTaskDetails,
17.     Date newEditDate, ArrayList newSubTasks){
18.       lastEditDate = newEditDate;
19.       taskName = newTaskName;
20.       taskDetails = newTaskDetails;
21.      if (newSubTasks != null){ subTasks = newSubTasks; }
22.  }
23.
24.  public String getTaskID(){
25.     return taskID;
26.  }
27.  public Date getLastEditDate(){ return lastEditDate; }
28.  public String getTaskName(){ return taskName; }
29.  public String getTaskDetails(){ return taskDetails; }
30.  public ArrayList getSubTasks(){ return subTasks; }
31.
32.  public void setLastEditDate(Date newDate){
33.     if (newDate.after(lastEditDate)){
```

```
34.      lastEditDate = newDate;
35.    }
36.  }
37.  public void setTaskName(String newName){ taskName = newName; }
38.  public void setTaskDetails(String newDetails){ taskDetails = newDetails; }
39.  public void addSubTask(Task task){
40.    if (!subTasks.contains(task)){
41.      subTasks.add(task);
42.    }
43.  }
44.  public void removeSubTask(Task task){
45.    subTasks.remove(task);
46.  }
47.
48.  public String toString(){
49.    return taskName + " " + taskDetails;
50.  }
51.}
```

Листинг А.242. UpdateException.java

```
1. public class UpdateException extends Exception{
2.   public static final int TASK_UNCHANGED = 1;
3.   public static final int TASK_OUT_OF_DATE = 2;
4.   private int errorCode;
5.
6.   public UpdateException(String cause, int newErrorCode){
7.     super(cause);
8.     errorCode = newErrorCode;
9.   }
10.  public UpdateException(String cause){ super(cause); }
11.
12.  public int getErrorCode(){ return errorCode; }
13.}
```

Демонстрация того, как обновление данных некоторого этапа проекта может автоматически доставляться нескольким клиентам, осуществляется классом RunPattern (листинг А.243). Метод main этого класса создает сервер, представленный экземпляром класса ClientPullServer, а также двух клиентов, представленных экземплярами класса PullClient. Оба клиента используются для отправки запросов на получение обновлений одного и того же этапа проекта, после чего один из клиентов выполняет обновление этого этапа. Внесенные изменения будут отражены вторым клиентом, как только его обработчик потока (экземпляр класса ClientPullRequester) в очередной раз запросит сведения об обновлениях у сервера.

Листинг А.243. RunPattern.java

```
1. import java.io.IOException;
2. public class RunPattern{
3.   public static void main(String [] arguments){
4.     System.out.println("Example for the SuccessiveUpdate pattern");
5.     System.out.println("This code provides a basic demonstration");
6.     System.out.println(" of how the client pull form of this pattern");
7.     System.out.println(" could be applied.");
8.     System.out.println("In this case, a change made by a client to a");
```

```
9.    System.out.println(" central Task object is subsequently retrieved");
10.   System.out.println(" and displayed by another client.");
11.
12.   System.out.println("Running the RMI compiler (rmic)");
13.    System.out.println();
14.    try{
15.     Process p1 - Runtime.getRuntime().exec("rmic ClientPullServerImpl");
16.      p1.waitFor();
17.    )
18.   catch (IOException exc) {
19.    System.err.println("Unable to run rmic utility. Exiting application.");
20.      System.exit(1);
21.    }
22.   catch (InterruptedException exc) (
23.     System.err.println("Threading problems encountered while using the
   rmic utility.");
24.    }
25.
26.    System.out.println("Starting the rmiregistry");
27.    System.out.println();
28.   Process rmiProcess = null;
29.    try{
30.      rmiProcess = Runtime.getRuntime().exec("rmiregistry");
31.      Thread.sleep(15000);
32.    }
33.   catch (IOException exc) {
34.     System.err.println("Unable to start the rmiregistry. Exiting
   application.");
35.     System.exit(1) ;
36.    }
37.    catch (InterruptedException exc){
38.     System.err.println("Threading problems encountered when starting the
   rmiregistry.");
39.    >
40.
41.   System.out.println("Creating the ClientPullServer and two PullClient
   objects") ;
42.   ClientPullServer server = new ClientPullServerImpl();
43.   PullClient clientOne = new PullClient("Thing I");
44.   PullClient clientTwo - new PullClient("Thing II");
45.   clientOne.requestTask("First work step");
46.   clientTwo.requestTask("First work step");
47.
48.    try{
49.      Thread.sleep(10000);
50.    )
51.   catch (InterruptedException exc){ }
52.
53.    Task task - clientOne.getUpdatedTask();
54.   task.setTaskDetails("Trial for task update");
55.   clientOne.updateTask(task) ;
56.
57.    Task newTask = clientTwo.getUpdatedTask();
58.   newTask.setTaskDetails("New details string");
59.   clientTwo.updateTask(newTask) ;
60.
61. }
62.}
```

Rotifer

Шаблон Router можно с успехом применять в нескольких местах рассматриваемого примера приложения. Практически в любой ситуации, когда имеется несколько получателей информации можно применить шаблон Router. По сути, маршрутизатор — это реализация структуры, подобной структуре объектов класса Listener.

В данном примере представлен исходный код класса Message (листинг А.244). Этот класс представляет собой контейнер для источника (InputChannel, листинг А.245) и собственно сообщение, которое здесь представлено переменной типа String.

Листинг А.244. Message.java

```
1. import java.io.Serializable;
2. public class Message implements Serializable{
3.   private InputChannel source;
4.   private String message;
5.
6.   public Message(InputChannel source, String message){
7.     this.source = source;
8.     this.message = message;
9.   }
10.
11.  public InputChannel getSource(){ return source; }
12.  public String getMessage(){ return message; }
13.}
```

Листинг А.245. InputChannel.java

```
1. import java.io.Serializable;
2. public interface InputChannel extends Serializable{}
```

Выходной канал OutputChannel (листинг А.246) — это интерфейс, который определяет метод отправки сообщения получателю. Поскольку выходной канал может использоваться для коммуникаций между разными машинами, он определен как удаленный интерфейс.

Листинг А.246. OutputChannel.java

```
1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface OutputChannel extends Remote{
4.   public void sendMessage(Message message) throws RemoteException;
5. }
```

Класс Router (листинг А.247) имеет хеш-таблицу, в которой хранятся связи, установленные между заданным входным каналом и различными выходными каналами. Когда экземпляр класса получает сообщение, он по этой таблице определяет получателя.

Экземпляр класса перебирает все элементы коллекции и отправляет сообщение каждому получателю. В данном примере класс Router для отправки сообщения каж-

дому объекту класса OutputChannel создает обработчик потока (подробнее см. раздел "Worker Thread" на стр. 507). Для повышения эффективности подобных приложений часто используются пулы потоков.

Листинг А.247. Router.java

```
1. import java.rmi.Naming;
2. import java.rmi.RemoteException;
3. import java.rmi.server.UnicastRemoteObject;
4. import java.util.HashMap;
5. public class Router implements OutputChannel{
6.   private static final String ROUTER_SERVICE_NAME = "router";
7.   private HashMap links = new HashMap();
8.
9.   public Router (){
10.    try{
11.       UnicastRemoteObject.exportObject(this);
12.       Naming.rebind(ROUTER_SERVICENAME, this);
13.    }
14.    catch (Exception exc){
15.      System.err.println("Error using RMI to register the Router " + exc);
16.    }
17.  }
18.
19.  public synchronized void sendMessage(Message message) {
20.    Object key = message.getSource();
21.    OutputChannel[] destinations = (OutputChannel[])links.get(key);
22.    new RouterWorkThread(message, destinations);
23.  }
24.
25.  public void addRoute(InputChannel source, OutputChannel[] destinations) {
26.    links.put(source, destinations);
27.  }
28.
29.  private class RouterWorkThread implements Runnable{
30.    private OutputChannel [] destinations;
31.     private Message message;
32.     private Thread runner;
33.
34.     private RouterWorkThread(Message newMessage, OutputChannel[] newDestinations) {
35.       message = newMessage;
36.       destinations = newDestinations;
37.       runner = new Thread(this);
38.        runner.start();
39.     }
40.
41.     public void run() {
42.       for (int i = 0; i < destinations.length; i++){
43.           try{
44.                destinations[i].sendMessage(message);
45.            }
46.            catch(RemoteException exc){
47.                System.err.println("Unable to send message to " +
   destinations [i]);
48.            }
49.       }
50.     }
51. }
52.}
```

При использовании шаблона Router требуется обращать внимание на размер посылаемых сообщений. Как правило, сообщение должно быть максимально кратким. Однако при работе с объектами языка Java легко упустить из вида одну особенность. Объект может ссылаться на другие объекты, которые в свою очередь могут ссылаться на другие объекты и т.д. Поэтому пересылка небольшого, на первый взгляд, объекта может на самом деле повлечь за собой пересылку огромного объема информации. Например, отправка объекта `java.awt.Button` — это не очень хорошая идея, поскольку произойдет сериализация и отправка всего графического интерфейса.

Это очень похоже на то, чем может обернуться покупка детской игрушки. На первый взгляд может показаться, что приобретение трансформера — это не очень большая трата, но со временем, оценив, во что обходятся вам все аксессуары (дополнительный протектор, лазерный меч и т.п.), вы поневоле задумаетесь, не дешевле ли было купить ребенку красивый свитер.

В данном примере интерфейс InputChannel реализуется классом InputKey (листинг А.248). Поскольку экземпляры этого класса пересылаются маршрутизатору с использованием технологии RMI, данный класс должен содержать определения методов hashCode и equals, чтобы обеспечить сравнение объектов, выполняющихся на разных виртуальных машинах Java (JVM).

Листинг А.248. InputKey.java

```
1. public class InputKey implements InputChannel{
2.   private static int nextValue = 1;
3.   private int hashVal = nextValue++;
4.   public int hashCode(){ return hashVal; }
5.   public boolean equals(Object object){
6.     if (!(object instanceof InputKey)){ return false; }
7.     if (object.hashCode() != hashCode()){ return false; }
8.       return true;
9.   }
10.}
```

Создание клиента объекта класса Router обеспечивается классом RouterClient (листинг А.249). Этот класс может как отправлять, так и получать сообщения по технологии RMI. Метод sendMessageToRouter передает сообщения центральному маршрутизатору, а метод sendMessage (определен в интерфейсе OutputChannel, см. листинг А.246) — получает их от экземпляра класса Router.

Листинг А.249. RouterClient.java

```
1. import java.rmi.Naming;
2. import java.rmi.server.UnicastRemoteObject;
3. import java.rmi.RemoteException;
4. public class RouterClient implements OutputChannel{
5.   private static final String ROUTER_CLIENT_SERVICE_PREFIX = "routerClient";
6.   private static final String ROUTER_SERVER_MACHINE_NAME = "localhost";
7.   private static final String ROUTER_SERVER_SERVICE_NAME = "router";
8.   private static int clientIndex = 1;
9.   private String routerClientServiceName = ROUTER_CLIENT_SERVICE_PREFIX +
   clientIndex++;
10.  private OutputChannel router;
11.  private Receiver receiver;
```

```
    public RouterClient(Receiver receiver){
        this.receiver = receiver;
        try {
            UnicastRemoteObject.exportObject(this);
            Naming.rebind(routerClientServiceName, this);
            String url = "//" + ROUTER_SERVER_MACHINE_NAME + "/" +
ROUTER_SERVER_SERVICE_NAME;
            router = (OutputChannel)Naming.lookup(url);
        }
        catch (Exception exc){
            System.err.println("Error using RMI to register the Router
        }

    }
    public void sendMessageToRouter(Message message){
        try {
            router.sendMessage(message);
        }
        catch (RemoteException exc){}
    }
    public void sendMessage(Message message){
        receiver.receiveMessage(message);
    }
    public String toString(){
        return routerClientServiceName;
    }
}
```

Каждый экземпляр класса RouterClient обменивается данными с клиентом, представленным классом RouterGui (листинг А.251). Этот класс обеспечивает простой графический пользовательский интерфейс для обиена сообщениями с экземпляром класса Router. Класс RouterGui реализует интерфейс Receiver (листинг А.250), который позволяет экземпляру класса RouterClient получать обновления в режиме реального времени при поступлении сообщений от маршрутизатора.

Листинг А.250. Receiver.java

```
public interface Receiver{
    public void receiveMessage(Message message);
}
```

Листинг А.251. RouterGui.java

```
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JTextArea;
```

```
10.import javax.swing.JScrollPane;
11.import javax.swing.JTextField;
12.import javax.swing.JLabel;
13.import javax.swing.JPanel;
14.import java.io.Serializable;
15.public class RouterGui implements ActionListener, Receiver{
16. private static int instanceCount = 1;
17. private RouterClient routerClient;
18. private JFrame mainFrame;
19. private JButton exit, clearDisplay, sendMessage;
20. private JTextArea display;
21. private JTextField inputTextField;
22. private InputChannel inputChannel;
23.
24. public OutputChannel getOutputChannel(){
25.    return routerClient;
26. }
27.
28. public RouterGui(InputChannel newInputChannel){
29.    inputChannel = newInputChannel;
30.    routerClient = new RouterClient(this);
31. }
32.
33. public void createGui(){
34.    mainFrame = new JFrame("Demonstration for the Router pattern - GUI #" +
    instanceCount++);
35.    Container content = mainFrame.getContentPane();
36.    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
37.
38.    JPanel displayPanel = new JPanel();
39.    display = new JTextArea(10, 40);
40.    JScrollPane displayArea = new JScrollPane(display);
41.    display.setEditable(false);
42.    displayPanel.add(displayArea);
43.    content.add(displayPanel);
44.
45.    JPanel dataPanel = new JPanel();
46.    dataPanel.add(new JLabel("Message:"));
47.    inputTextField = new JTextField(30);
48.    dataPanel.add(inputTextField);
49.    content.add(dataPanel);
50.
51.    JPanel controlPanel = new JPanel();
52.    sendMessage = new JButton("Send Message");
53.    clearDisplay = new JButton("Clear");
54.    exit = new JButton("Exit");
55.    controlPanel.add(sendMessage);
56.    controlPanel.add(clearDisplay);
57.    controlPanel.add(exit);
58.    content.add(controlPanel);
59.
60.    sendMessage.addActionListener(this);
61.    clearDisplay.addActionListener(this);
62.    exit.addActionListener(this);
63.    inputTextField.addActionListener(this);
64.
65.    mainFrame.addWindowListener(new WindowCloseManager());
66.    mainFrame.pack();
67.    mainFrame.setVisible(true);
68. }
69.
70. public void actionPerformed(ActionEvent evt){
71.    Object source = evt.getSource();
```

```
72.    if (source == sendMessage){ sendMessage(); }
73.   else if (source == inputTextField)) sendMessage(); }
74.   else if (source == clearDisplay){ clearDisplay (); }
75.   else if (source == exit){ exitApplication(); }
76.  )
77.
78. private class WindowCloseManager extends WindowAdapter{
79.    public void windowClosing(WindowEvent evt){
80.      exitApplication();
81.    }
82. }
83.
84. private void exitApplication(){
85.    System.exit(0);
86. }
87.
88. private void clearDisplay(){
89.    inputTextField.setText("");
90.    display.setText("");
91. }
92.
93. private void sendMessage(){
94.    String data = inputTextField.getText();
95.   routerClient.sendMessageToRouter(new Message(inputChannel, data));
96.    inputTextField.setText("");
97. }
98.
99. public void receiveMessage(Message message){
100.  display.append(message.getMessage() + "\n");
101. }
102.}
```

Класс `RunPattern` (листинг А.252) координирует работу всех классов, входящих в данный пример, создавая при этом серию объектов класса `RouterGui`. В рассматриваемом примере каждый такой объект подключается к нескольким другим подобным объектам через экземпляр класса Router. При этом обеспечивается, чтобы сообщение, отправленное четвертым объектом класса RouterGui, было доставлено всем остальным объектам данного класса, а сообщение, отправленное первым объектом, было доставлено лишь второму и третьему объектам.

Листинг А.252. RunPattern.java

```
1. import java.io.IOException;
2. public class RunPattern)
3.  public static void main(String [] arguments))
4.     System.out.println("Example for the Router pattern");
5.    System.out.println("This code same will create a series of GUIs, and use");
6.     System.out.println(" the Router pattern to map message notifications
   between");
7.     System.out.println(" them. In this code example, the Router will send
   messages");
8.    System.out.println (" between the GUI clients based on the following
   mapping:");
9.     System.out.println();
10.    System.out.println("\tGUI# 1:\tGUI #2\tGUI #3");
11.    System.out.println("\tGUI # 2:\tGUI #1\tGUI #4");
12.    System.out.println("\tGUI # 3:\tGUI #1\tGUI #4");
13.    System.out.println("\tGUI # 4:\tGUI #1\tGUI #2\tGUI #3\tGUI #4");
14.    System.out.println();
```

```
15.
16.    System.out.println("Running the RMI compiler (rmic)");
17.    try{
18.      Process p1 = Runtime.getRuntime().exec("rmic Router");
19.      Process p2 = Runtime.getRuntime().exec("rmic RouterClient");
20.      p1.waitFor();
21.      p2.waitFor();
22.    }
23.    catch (IOException exc) {
24.      System.err.println("Unable to run rmic utility. Exiting application.");
25.      System.exit (1);
26.    }
27.    catch (InterruptedException exc){
28.      System.err.println("Threading problems encountered while using the
   rmic utility.");
29.    }
30.
31.
32.    System.out.println("Starting the rmiregistry");
33.    System.out.println();
34.    Process rmiProcess = null;
35.    try{
36.     rmiProcess = Runtime.getRuntime().exec("rmiregistry");
37.      Thread.sleep(15000);
38.        }

          System.err.println("Unable

        System.exit

          System.err.println("Threading



       System.out.println("Creating    Router object");
48.    System.out.println();
49.    Router mainRouter = new Router();
50.
51.    InputKey keyOne = new InputKey();
52.    InputKey keyTwo = new InputKey();
53.    InputKey keyThree = new InputKey();
54.    InputKey keyFour = new InputKey();
55.
56.    System.out.println("Creating the four RouterGui objects");
57.    System.out.println();
58.    RouterGui first = new RouterGui(keyOne);
59.    RouterGui second = new RouterGui(keyTwo);
60.    RouterGui third = new RouterGui(keyThree);
61.    RouterGui fourth = new RouterGui(keyFour);
62.
63.    System.out.println("Creating GUI OutputChannel lists for the Router");
64.  System.out.println();
65.    OutputChannel [] subscriptionListOne = { second.getOutputChannel(),
   third.getOutputChannel() };
66.    OutputChannel [] subscriptionListTwo = { first.getOutputChannel(),
   fourth.getOutputChannel() };
67.    OutputChannel [] subscriptionListThree = { first.getOutputChannel(),
   second.getOutputChannel(),
68.      third.getOutputChannel(), fourth.getOutputChannel() };
69.
70.    mainRouter.addRoute(keyOne, subscriptionListOne);
71.    mainRouter.addRoute(keyTwo, subscriptionListTwo);
```

```
72.    mainRouter.addRoute(keyThree, subscriptionListTwo);
73.   mainRouter.addRoute(keyFour, subscriptionListThree) ;
74.
75.    first.createGui() ;
76.   second. createGui () ;
77.    third.createGui();
78.    fourth.createGui();
79.  }
80.}
```

Transaction

PIM-приложение сохраняет информацию о запланированных событиях, основываясь на дате их проведения. Естественно, поскольку пользователи ведут активный образ жизни, в их планах постоянно происходят изменения. Поэтому еженедельник пользователя постоянно обновляется для отражения новых планов или внесения изменений в уже намеченные планы.

Когда нескольким пользователям требуется согласовать дату проведения какого-то мероприятия, было бы неплохо, чтобы такое согласование выполнялось автоматически их еженедельниками, которые бы выбрали дату, устраивающую всех пользователей. Именно такая задача и решается в данном примере— с помощью шаблона Transaction создается механизм согласования даты проведения мероприятия, встраиваемый в подсистему **PIM-приложения**, которая выполняет функции еженедельника.

Базовым интерфейсом, обеспечивающим поддержку транзакций, является `AppointmentTransactionParticipant` (листинг А.253). Он содержит определение трех методов, обеспечивающих управление транзакцией (join, commit и cancel), а также одного прикладного метода changeDate. Этот интерфейс расширяет интерфейс Remote, поскольку он используется для организации обмена информацией между участниками транзакции, которые в общем случае могут выполняться в разных экземплярах виртуальной машины Java.

Листинг А.253. AppointmentTransactionParticipant.java

```
1. import java.util.Date;
2. import java.rmi.Remote;
3. import java.rmi.RemoteException;
4. public interface AppointmentTransactionParticipant extends Remote{
5.   public boolean join(long transactionID) throws RemoteException;
6.   public void commit(long transactionID) throws TransactionException,
   RemoteException;
7.   public void cancel(long transactionID) throws RemoteException;
8.   public boolean changeDate(long transactionID, Appointment appointment,
9.     Date newStartDate) throws TransactionException, RemoteException;
10.}
```

Класс `AppointmentBook` (листинг А.254), реализующий интерфейс AppointmentTransactionParticipant, представляет в приложении еженедельник пользователя. Помимо обеспечения изменения даты запланированного события, класс AppointmentBook может инициировать изменение самого события. Метод changeAppointment этого класса получает в качестве параметров: идентификатор транзак-

ции; объект Appointment; массив ссылок на другие экземпляры класса AppointmentBook, участвующие в транзакции; массив возможных дат проведения запланированного мероприятия. Метод changeAppointment позволяет одному из объектов класса AppointmentBook обмениваться информацией с остальными объектами по технологии RMI, вызывая метод changeDate каждого из участников транзакции до тех пор, пока не будет согласована дата проведения запланированного мероприятия.

Листинг А.254. AppointmentBook.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Date;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class AppointmentBook implements AppointmentTransactionParticipant{
private static final String TRANSACTION_SERVICE_PREFIX =
"transactionParticipant";
private static final String TRANSACTION_HOSTNAME = "localhost";
private static int index = 1;
private String serviceName = TRANSACTION_SERVICE_PREFIX + index++;
private HashMap appointments = new HashMap();
private long currentTransaction;
private Appointment currentAppointment;
private Date updateStartDate;

public AppointmentBook(){
  try {
    UnicastRemoteObject.exportObject(this);
    Naming.rebind(serviceName, this);
  }
  catch (Exception exc){
  System.err.println("Error using RMI to register the AppointmentBook "
+ exc);
  }
}

public String getUrl(){
  return "//" + TRANSACTION_HOSTNAME + "/" + serviceName;
}

public void addAppointment(Appointment appointment){
  if (!appointments.containsValue(appointment)){
if (!appointments.containsKey(appointment.getStartDate())){
    appointments.put(appointment.getStartDate(), appointment);
    }
  }
}
public void removeAppointment(Appointment appointment){
  if (appointments.containsValue(appointment)){
    appointments.remove(appointment.getStartDate());
  }
}

public boolean join(long transactionID){
  if (currentTransaction != 0){
    return false;
  } else {
    currentTransaction = transactionID;
    return true;
```

```
50.   ,}
51.  }
52.  public void commit(long transactionXD) throws TransactionException{
53.     if (currentTransaction !- transactionID){
54.       throw new TransactionException("Invalid TransactionID");
55.     } else {
56.       removeAppointment(currentAppointment);
57.       currentAppointment.setStartDate(updateStartDate);
58.      appointments.put(updateStartDate, currentAppointment);
59.     )
60.  )
61.  public void cancel(long transactionID){
62.     if (currentTransaction == transactionID){
63.       currentTransaction - 0;
64.       appointments.remove(updateStartDate);
65.     }
66.  }
67.  public boolean changeDate(long transactionID, Appointment appointment,
68.    Date newStartDate) throws TransactionException{
69.      if ((appointments.containsValue(appointment)) &&
   (!appointments.containsKey(newStartDate))){
70.           appointments.put(newStartDate, null);
71.           updateStartDate - newStartDate;
72.           currentAppointment « appointment;
73.           return true;
74.       }          ;
75.     return false;
76.  }
77.
78.  public boolean changeAppointment(Appointment appointment, Date[]
   possibleDates,
79.  AppointmentTransactionParticipant[] participants, long transactionID){
80.    try{
81.      for (int i = 0; i<,participants.length; i++){
82.           if (!participants[i].join(transactionID)){
83.              return false»
84.       }
85.     }
86.     for (int ,i = 0; i < possibleDates.length; i++) {
87.          if (isDateAvailable(transactionID, appointment,
   possibleDates[i] ,participants)) {
88.            try{
89.                    commitAll(transactionID, participants);
90.                    return true;
                   }
92.                catch(TransactionException exc){ }
93.           }
94.       }
95.     )
96.     catch (RemoteException exc){ }
97.   try{
98.       cancelAll(transactionID, participants);
99.     )
100.     catch (RemoteException ёxc){}
101.     return false;
102.  }
103.
104.  private boolean isDateAvailable(long transactionID, Appointment appointment,
105.  Date date, AppointmentTransactionParticipant[] participants){
106.     try<
107.      for (int i = 0; i < participants.length; i++) (
108.           try{
```

```
109.            if (!participants[i].changeDate(transactionID, appointment,
    date)){
110.                return false;
111.
112.            }
113.            catch (TransactionException exc){
114.                return false;
115.            }
116.        }
117.        catch (RemoteException exc){
118.            return false;
119.        }
120.    }
121.    return true;
122. }
123. private void commitAll(long transactionID,
    AppointmentTransactionParticipant[] participants)
124.     throws TransactionException, RemoteException{
125.     for (int i = 0; i < participants.length; i++){
126.         participants[i].commit(transactionID);
127.     }
128. }
129. private void cancelAll(long transactionID,
    AppointmentTransactionParticipant[] participants)
130.     throws RemoteException{
131.     for (int i = 0; i < participants.length; i++){
132.         participants[i].cancel(transactionID);
133.     }
134. }
135. public String toString(){
136.     return serviceName + " " + appointments.values().toString();
137. }
138.}
```

Класс TransactionException (листинг А.255) представлят собой сигнальную исключительную ситуацию, поскольку не имеет какого-либо специального содержимого. Соответствующая исключительная **ситуация** генерируется некоторыми методами при получении некорректной транзакции. **Получатель** такой транзакции может либо известить о возникновении исключительной ситуации, либо проигнорировать ее, в зависимости от того, как в каждом конкретном случае удобнее организовать работу.

Листинг А.255. TransactionException.java

```
1. public class TransactionException extends Exception{
2.   public TransactionException(String msg){
3.     super(msg);
4.   }
5. }
```

Вспомогательные классы в данном примере представляют запланированные события и их элементы. Поведение **соответсвующих** объектов определяется содержимым трех интерфейсов; Appointment (листинг А. 256), Contact (листинг А.258) и Location (листинг **А.260).** Реализация **поведения** осуществляется классами **AppointmentImpl** (листинг А. 257), **ContactImpl** (листинг **А.259)** и **LocationImpl (листинг А.261).**

Листинг А.256. Appointment.java

```
1. import java.util.ArrayList;
2. import java.util.Date;
3. import java.io.Serializable;
4. public interface Appointment extends Serializable{
5.   public static final String EOL_STRING =
   System.getProperty("line.separator");
6.
7.   public Date getStartDate();
8.   public String getDescription();
9.   public ArrayList getAttendees();
10.  public Location getLocation();
11.
12.  public void setDescription(String newDescription);
13.  public void setLocation(Location newLocation);
14.  public void setStartDate(Date newStartDate);
15.  public void setAttendees(ArrayList newAttendees);
16.  public void addAttendee(Contact attendee);
17.  public void removeAttendee(Contact attendee);
18.}
```

Листинг А.257. AppointmentImpl.java

```
1. import java.util.ArrayList; . .
2. import java.util.Date;
3. public class AppointmentImpl implements Appointment{
4.   private Date startDate;
5.   private String description;
6.   private ArrayList attendees = new ArrayList();
7.   private Location location;
8.
9.   public AppointmentImpl(String newDescription, ArrayList newAttendees,
10.     Location newLocation, Date newStartDate){
11.       description = newDescription;
12.       attendees = newAttendees;
13.       location = newLocation;
14.       startDate = newStartDate;
15.  >
16.
17.  public Date getStartDate(){ return startDate; }
18.  public String getDescription(){ return description; }
19.  public ArrayList getAttendees(){ return attendees; }
20.  public Location getLocation(){ return location; )
21.
22.  public void setDescription(String newDescription); description =
   newDescription; )
23.  public void setLocation(Location newLocation){ location = newLocation; }
24.  public void setStartDate(Date newStartDate)( startDate = newStartDate; }
25.  public void setAttendees(ArrayList newAttendees){
26.     if (newAttendees != null)(
27.       attendees = newAttendees;
28.     }
29.  }
30.
31.  public void addAttendee(Contact attendee){
32.     if (!attendees.contains(attendee)){
33.       attendees.add(attendee);
34.     }
35.  }
```

```
36.
37.  public void removeAttendee(Contact attendee){
38.     attendees.remove(attendee);
39.  }
40.
41.  public int hashCode(){
42.     return description.hashCode() ^ startDate.hashCode();
43.  }
44.
45.  public boolean equals(Object object){
46.     if (!(object instanceof AppointmentImpl)){
47.       return false;
48.     }
49.     if (object.hashCode() != hashCode()){
50.       return false;
51.     }
52.     return true;
53.  }
54.
55.  public String toString(){
56.     return " Description: " + description + EOL_STRING +
57.       " Start Date: " + startDate + EOL_STRING +
58.       " Location: " + location + EOL_STRING +
59.       " Attendees: " + attendees;
60.  }
61.}
```

Листинг А.258. Contact.java

```
1. import java.io.Serializable;
2. public interface Contact extends Serializable{
3.    public static final String SPACE = " ";
4.    public String getFirstName();
5.    public String getLastName();
6.    public String getTitle();
7.    public String getOrganization();
8.
9.    public void setFirstName(String newFirstName);
10.   public void setLastName(String newLastName);
11.   public void setTitle(String newTitle);
12.   public void setOrganization(String newOrganization);
13.}
```

Листинг А.259. ContactImpl.java

```
1. public class ContactImpl implements Contact{
2.    private String firstName;
3.    private String lastName;
4.    private String title;
5.    private String organization;
6.
7.    public ContactImpl(){}
8.    public ContactImpl(String newFirstName, String newLastName,
9.       String newTitle, String newOrganization){
10.        firstName = newFirstName;
11.        lastName = newLastName;
12.        title = newTitle;
13.        organization = newOrganization;
```

```
14. }
15.
16. public String getFirstName(){ return firstName; }
17. public String getLastName () { return lastName; }
18. public String getTitle(){ return title; }
19. public String getOrganization(){ return organization; }
20.
21. public void setFirstName(String newFirstName){ firstName = newFirstName; }
22. public void setLastName(String newLastName){ lastName = newLastName; }
23. public void setTitle (String newTitle) { title = newTitle; }
24. public void setOrganization(String newOrganization){ organization =
    newOrganization; }
25.
26. public String toString(){
27.    return firstName + SPACE + lastName;
28. }
29.}
```

Листинг А.260. Location.java

```
1. import java.io.Serializable;
2. public interface Location extends Serializable{
3.   public String getLocation();
4.   public void setLocation(String newLocation) ;
5. }
```

Листинг А.261. LocationImpl.java

```
1. public class LocationImpl implements Location{
2.   private String location;
3.
4.   public LocationImpl(){ }
5.   public LocationImpl(String newLocation){
6.      location = newLocation;
7.   }
8.
9.   public String getLocation(){ return location; }
10.
11.  public void setLocation(String newLocation){ location = newLocation; }
12.
13.  public String toString(){ return location; }
14.}
```

Класс `RunPattern` (листинг А.262) демонстрирует, как можно скоординировать работу нескольких еженедельников при изменении в данных запланированного события. Этот класс создает три экземпляра класса `AppointmentBook`, устанавливая в двух из них противоречащие друг другу данные об одном и том же запланированном событии. Затем он дает указание одному из объектов еженедельников обновить данные по этому событию. В результате во всех трех еженедельниках устанавливается единое время проведения данного события— 12 часов дня, поскольку именно на это время приходится первое "окно" в расписании всех трех пользователей.

Листинг А.262. RunPattern.java

```
1. import java.io.IOException;
2. import java.rmi.Naming;
3. import java.util.Date;
4. import java.util.Calendar;
5. import java.util.ArrayList;
6. public class RunPattern{
7.   private static Calendar dateCreator = Calendar.getInstance();
8.
9.   public static void main(String [] arguments){
10.    System.out.println("Example for the Transaction pattern");
11.    System.out.println("This code example shows how a Transaction can");
12.    System.out.println(" be applied to support change across a distributed");
13.    System.out.println(" system. In ths case, a distributed transaction");
14.    System.out.println(" is used to coordinate the change of dates in");
15.    System.out.println(" appointment books.");
16.
17.    System.out.println("Running the RMI compiler (rmic)");
18.    System.out.println();
19.    try{
20.      Process p1 = Runtime.getRuntime().exec("rmic AppointmentBook");
21.      p1.waitFor();
22.    }
23.    catch (IOException exc){
24.      System.err.println("Unable to run rmic utility. Exiting application.");
25.      System.exit(1);
26.    )
27.    catch (InterruptedException exc){
28.      System.err.println("Threading problems encountered while using the
   rmic utility.");
29.    }
30.
31.    System.out.println("Starting the rmiregistry");
32.    System.out.println();
33.    try{
34.      Process rmiProcess = Runtime.getRuntime().exec("rmiregistry");
35.      Thread.sleep(15000);
36.    }
37.    catch (IOException exc) {
38.      System.err.println("Unable to start the rmiregistry. Exiting application.");
39.      System.exit(1);
40.    >
41.    catch (InterruptedException exc){
42.      System.err.println("Threading problems encountered when starting the
   rmiregistry.");
43.    )
44.
45.    System.out.println("Creating three appointment books");
46.    System.out.println();
47.    AppointmentBook apptBookOne = new AppointmentBook();
48.    AppointmentBook apptBookTwo - new AppointmentBook();
49.    AppointmentBook apptBookThree = new AppointmentBook();
50.
51.    System.out.println("Creating appointments");
52.    System.out.println();
53.    Appointment apptOne = new AppointmentImpl("Swim relay to Kalimantan (or
   Java)", new ArrayList(),
54.      new LocationImpl("Sidney, Australia"), createDate(2001, 11, 5, 11, 0));
55.    Appointment apptTwo - new AppointmentImpl("Conference on World
   Patternization", new ArrayList(),
56.      new LocationImpl("London, England"), createDate(2001, 11, 5, 14, 0) ) ;
```

```
57.  Appointment apptThree = new AppointmentImpl("Society for the
   Preservation of Java - Annual Outing",
58.    new ArrayList(), new LocationImpl ("Kyzyl, Tuva"), createDate (2001, 11,
   5, 10, 0));
59.
60.    System.out.println("Adding appointments to the appointment books");
61.    System.out.println();
62.     apptBookOne.addAppointment(apptThree);
63.     apptBookTwo.addAppointment(apptOne);
64.    apptBookOne.addAppointment(apptTwo) ;
65.     apptBookTwo.addAppointment(apptTwo);
66.    apptBookThree.addAppointment(apptTwo) ;
67.
68.     System.out.println("AppointmentBook contents:");
69.     System.out.println();
70.    System.out.println(apptBookOne) ;
71.    System.out.println(apptBookTwo) ;
72.     System.out.println(apptBookThree);
73.     System.out.println();
74.
75.     System.out.println("Rescheduling an appointment");
76.     System.out.println();
77.     System.out.println();
78.     boolean result = apptBookThree.changeAppointment(apptTwo, getDates(2001,
   11, 5, 10, 3),
79.       lookUpParticipants(new String!] { apptBookOne.getUrl(),
   apptBookTwo.getUrl(), apptBookThree.getUrl() }),
80.       20000L);
81.
82.     System.out.println("Result of rescheduling was " + result);
83.     System.out.println("AppointmentBook contents:");
84.     System.out.println();
85.     System.out.println(apptBookOne);
86.     System.out.println(apptBookTwo);
87.     System.out.println(apptBookThree);
88.  }
89.
90.  private static AppointmentTransactionParticipant[]
   lookUpParticipants(String[] remoteUrls){
91.     AppointmentTransactionParticipant[] returnValues =
92.       new AppointmentTransactionParticipant[remoteUrls.length];
93.     for (int i - 0; i < remoteUrls.length; i++){
94.       try{
95.            returnValues[i] -
    (AppointmentTransactionParticipant)Naming.lookup(remoteUrls[i]);
96.       }
97.       catch (Exception exc){
98.            System.out.println("Error using RMI to look up a transaction
   participant");
99.       }
100.    }
101.    return returnValues;
102. }
103.
104. private static Date[] getDates(int year, int month, int day, int hour, int
   increment){
105.    Date[] returnDates = new Date[increment];
106.    for (int i = 0; i < increment; i++){
107.     returnDates[i] = createDate(year, month, day, hour + i, 0);
108.    }
109.    return returnDates;
110. }
111.
```

```
112. public static Date createDate(int year, int month, int day, int hour,
   int minute){
113.   dateCreator.set(year, month, day, hour, minute);
114.    return dateCreator.getTime();
115. }
116.}
```

ИСТОЧНИКИ ИНФОРМАЦИИ

Приложение Б

В данном приложении приведены сведения об источниках информации по шаблонам, которые рассмотрены в книге (табл. Б.1), а также полный перечень источников, которые упоминаются в книге по ходу изложения материала (табл. Б.2).

Таблица Б.1. Источники информации о шаблонах, рассмотренных в книге

Производящие шаблоны		*Структурные шаблоны*	
Abstract Factory	GoF	Adapter	GoF
Builder	GoF	Bridge	GoF
Factory Method	GoF	Composite	GoF
Prototype	GoF	Decorator	GoF
Singleton	GoF	Facade	GoF
		Flyweight	GoF
Поведенческиешаблоны			
Chain of Responsibility	GoF	HOPP	Coplien
Command	GoF	Proxy	GoF
Interpreter	GoF		
		Системныешаблоны	
Iterator	GoF	MVC	LeaOO
Mediator	GoF	Session	LeaOO
Memento	GoF	Worker Thread	LeaOO
Observer	GoF	Callback	LeaOO
State	GoF	Successive Update	Нет
Strategy	GoF	Router	DPCS
Template Method	GoF	Transaction	LeaOO
Visitor	GoF		

Таблица Б.2. Перечень источников информации о шаблонах

CJ2EEP

Deepak Alur, John Crupi, Dan Malks
CoreJ2EE Patterns
Prentice Hall, 2001
ISBN 0-13-066586-X

BlochOl

Joshua Bloch
Effective Java, Programming Language Guide
Addison Wesley, 2001
ISBN 0-201-31005-8

Coplien

Jim O. Coplien, Douglas C. Schmidt (Editors)
Patterns Languages of Program Design
Addison Wesley, 1995
ISBN 0-201-60734-4

DPCS

Design Patterns for Communications Software
Linda Rising (Editor)
Cambridge University Press, 2000
ISBN 0-521-79040-9

Fowler00

Martin Fowler
UML Distilled, Second Edition
Addisor. Wesley, 2000
ISBN 0-201-65783-X

Фаулер М., Скотт К.
UML. Основы. 2-е издание
Символ-Плюс, 2002
ISBN 5-93286-032-4

GoF

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns, Elements of Reusable Object-Oriented Software
Addison Wesley, 1995
ISBN 0-201-63361-2

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.
Приемы объектно-ориентированного проектирования. Паттерны проектирования
Издательский дом "Питер", ДМК, 2001
ISBN 5-272-00355-1

LeaOO

Doug Lea
Concurrent Programming in Java, Second Edition
Addison Wesley, 2000
ISBN 0-201-31009-0

JLS

Bill Joy (Editor), Guy Steele, James Gosling, Gilad Bracha
The Java Language Specification, Second Edition
Addison Wesley, 2000
ISBN 0-201-31008-2

JBS

JavaBeans Specification 1.01
`http://java.sun.com/products/javabeans/docs/spec.html`
`http://www.sun.ru/win/java/products/JavaBeans/index.html`

J2EE00

Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, Eduardo Pelegri-Llopart, Larry Cable, Enterprise Team
Java 2 Platform, Enterprise Edition, Platform and Component Specification
Addison Wesley, 2000
ISBN 0-201-70456-0

JiniOl

Jim Waldo, the Jini Technology Team
The Jini Specifications, Second Edition
Addison Wesley, 2001
ISBN 0-201-72617-3

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

A

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

П

P

C

Т

У

Ф

Х

Ш

Я

Научно-популярное издание

Стивен Стелтинг, Олав Маасен

Применение шаблонов Java. Библиотека профессионала

Литературный редактор	*С.Г. Татаренко*
Верстка	*А.В. Плаксюк*
Художественный редактор	*В.Г. Павлютин*
Корректоры	*З.В. Александрова, Т.А. Корзун, О.В. Мишутина, Г.В. Павлюченко, И.Ф. Моторина, В.И. Ступак, Г.И. Шумак*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 14.08.2002. Формат 70x100/16.
Гарнитура NewBaskerville. Печать офсетная.
Усл. печ. л. 33,54. Уч.-изд. л. 30,00.
Тираж 3000 экз. Заказ № 1107.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

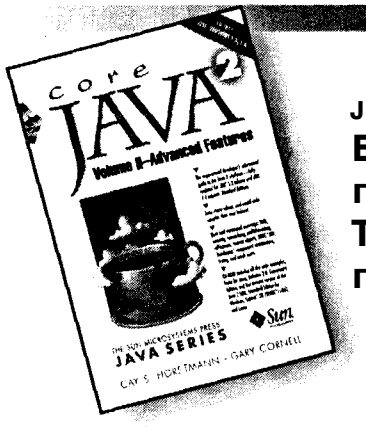
Кей Хорстманн, Гари Корнелл

Java 2. Библиотека профессионала. Том 1. Основы

Плановая дата выхода
4-й кв. 2002 г.

Книга адресована, прежде всего, программистам-профессионалам и представляет собой исчерпывающий справочник и методическое пособие по основам программирования на языке Java. Однако это не просто учебник по синтаксису языка. Назначение книги — обучить методам объектно-ориентированного программирования и решению основных проблем в этой области. Работа с книгой не требует предыдущего опыта программирования на языке C++ и применения методов ООП. Любой программист, работавший с языками Visual Basic, C, Cobol или Pascal, не будет испытывать затруднений при работе с ней. Книга содержит многочисленные примеры и советы по программированию, а также разделы, в которых рассматриваются методы тестирования и отладки программ, абстрактные типы данных, базовое объектно-ориентированное программирование, включающее событийно-управляемое программирование. Книгу можно использовать не только как учебник, но и как справочник.

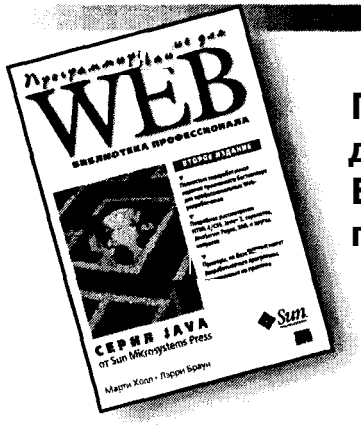
Кей Хорстманн, Гари Корнелл



Java 2. Библиотека профессионала. Том 2. Тонкости программирования

Плановая дата выхода 2-й кв. 2002 г.

Эта книга посвящена описанию более сложных вопросов профессионального программирования на Java и предназначена для тех программистов, которые хотят использовать технологию Java для создания реальных проектов. Главы книги в основном не связаны друг с другом, поэтому их можно читать независимо друг от друга и в любом порядке. В главе 1 рассматривается механизм многопоточности, который позволяет программировать параллельное выполнение разных задач. В главе 2 описываются коллекции, используемые в платформе Java 2. В главе 3 описывается один из наиболее важных API-интерфейсов платформы Java, который предназначен для работы с сетями. В главе 4 представлен JDBC —Java API-интерфейс для работы с базами данных. В главе 5 рассматриваются удаленные объекты и технология удаленных вызовов (Remote Method Invocation — RMI). В главе 6 содержится дополнительный материал о библиотеке Swing, который не удалось полностью разместить в томе I. В главе 7 рассматривается 2D API-интерфейс, который позволяет создавать реалистичные изображения. В главе 8 описывается API-интерфейс для работы с компонентами платформы Java — JavaBeans. В главе 9 рассматривается модель обеспечения безопасности Java. В главе 10 описываются инструменты локализации Java-приложений. В главе 11 рассматриваются встроенные методы, которые позволяют создавать методы для специальной платформы. В главе 12, которая появилась только в этом издании книги, рассматривается язык XML.



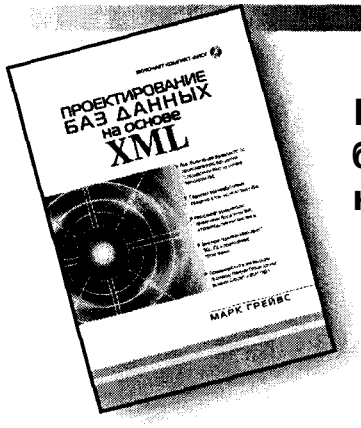
Марти Холл, Лэрри Браун

Программирование для Web. Библиотека профессионала

В продаже

В данной книге читатель найдет все необходимые сведения, позволяющие создавать Web-страницы, включать в них исполняемый код, а также реализовывать программы, выполняющиеся на стороне сервера. В ней достаточно полно описаны языковые конструкции, соответствующие спецификации HTML 4.0, приведены подробные сведения о языке Java, рассматривается создание программ, выполняющихся на стороне сервера (сервлетов и JSP), взаимодействие с базами данных, обработка XML-документов, построение программ, выполняющихся на стороне клиента, (аплетов и JavaScript-сценариев) и многие другие вопросы.

Марк Грейвс



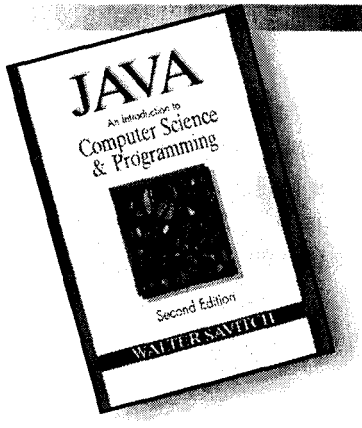
Проектирование баз данных на основе XML

В продаже

В этом всеобъемлющем руководстве рассматривается широкий круг задач проектирования баз данных XML, предназначенных для использования в среде Web и на производстве. Если в вашем распоряжении уже имеется система управления базой данных с поддержкой XML, то здесь вы найдете удобные методы проектирования, позволяющие добиться максимального повышения эффективности ее работы. Если вы работаете с обычной реляционной СУБД, то узнаете самые лучшие способы ее использования для разработки приложений XML. А если вы собираетесь создать базу данных на основе XML самостоятельно, то сможете овладеть всеми нужными для этого знаниями на практическом примере, в котором процесс построения такой базы данных описан от начала до конца.

Книга предназначена для пользователей средней и высокой квалификации.

Уолтер Савитч

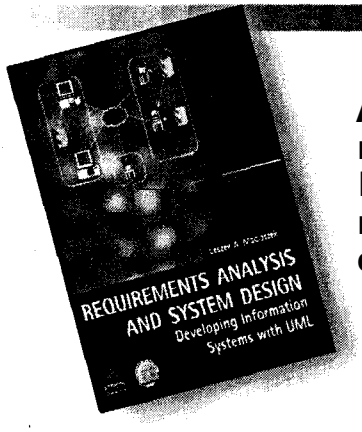


Язык Java. Курс программирования. 2-е изд.

Плановая дата выхода
2-й кв. 2002 г.

Книга адресована начинающим. Это не просто учебник по синтаксису языка Java, ее цель — обучить методам программирования и решению основных проблем в этой области. Работа с книгой не требует предыдущего опыта программирования и никакой особенной математической подготовки. Она содержит многочисленные примеры и советы по программированию, а также разделы, в которых рассматриваются методы тестирования и отладки программ, абстрактные типы данных, базовое объектно-ориентированное программирование, включающее событийно-управляемое программирование. Книгу можно использовать не только как учебник, но и как справочник.

Лешек Мацяшек



Анализ требований и проектирование систем. Разработка информационных систем с использованием UML

Плановая дата выхода — 2-й квартал 2002 г.

В книге описывается методология и технология объектно-ориентированной разработки современных информационных систем (ИС) и предлагается итеративный подход к разработке ИС с пошаговым наращиванием ее возможностей. Весь комплекс вопросов анализа и проектирования ИС рассматривается в контексте использования языка UML как универсального средства моделирования проектных решений.

Изложение ведется в соответствии с подходом, который можно назвать "обучением на примерах". Приведенные в книге примеры тщательно анализируются применительно к каждому из этапов создания ИС; доходчиво и убедительно демонстрируется путь преобразования неформальных требований заказчика в артефакты языка UML.

Отличительной чертой книги является гармоничное сочетание практического акцента в объяснении материала с глубоким проникновением в его теоретическую суть. Книга написана с позиций реального опыта.

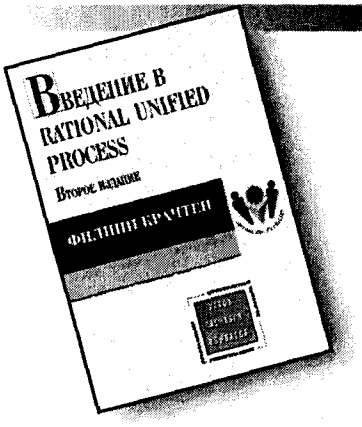
Книга предназначена для разработчиков ИС, кроме того, она может служить основой фундаментального курса обучения методам проектирования ИС и использования языка UML.

Инженерия программного обеспечения

Плановая дата выхода — 3-й квартал 2002 г.

Данная книга является прекрасным введением в инженерию программного обеспечения. Здесь дана широкая панорама тем инженерии ПО, охватывающих все этапы и технологии разработки программных систем. В семи частях книги представлен весь спектр процессов, ведущих к созданию программного обеспечения, начиная от начальной разработки системных требований и далее через проектирование, непосредственное программирование и аттестацию до модернизации программных систем. Эта книга окажет неоценимую поддержку студентам и аспирантам, изучающим дисциплину "Инженерия программного обеспечения", а также будет полезна тем специалистам по программному обеспечению, который хотят познакомиться с новыми технологиями разработки ПО, такими как спецификация требований, архитектура распределенных структур или надежность программных систем.

Ян Соммервилл



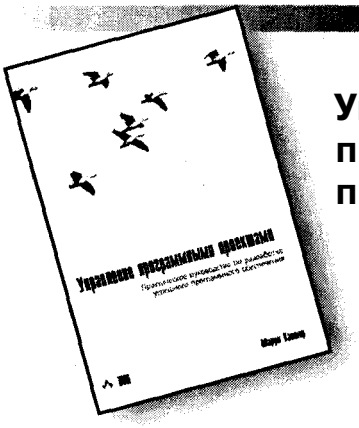
Введение в Rational Unified Process

2-е издание

В продаже

Технология Rational Unified Process - это то, как нужно разрабатывать программное обеспечение. Данная книга позволит вам подойти к этому процессу творчески и в то же время избежать характерных ошибок. Вы освоите новую терминологию и углубите имеющиеся знания, оцените предлагаемые советы и выработаете свой подход к созданию качественного (и не очень дорогого) программного обеспечения. Данную работу можно использовать и как руководство по разработке программного обеспечения, и просто как полезный материал для всех тех, кто связан с его разработкой. Данная книга также позволяет оценить все преимущества языка UML, хотя для ее прочтения совсем не обязательны фундаментальные знания о нем.

Марри Кантор

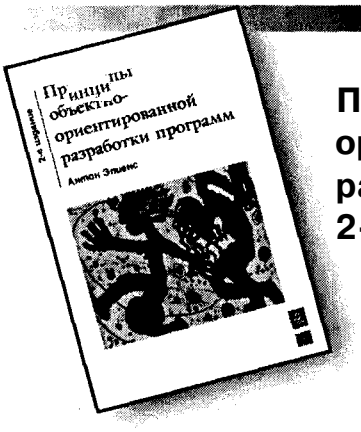


Управление программными проектами

В продаже

Разработка программных продуктов в силу своей специфики требует гораздо большего внимания и усилий, чем выполнение любого другого проекта. Эта книга поможет глубже заглянуть в сущность программных продуктов, процесса их разработки и используемых для этого инструментов и средств. Прочитав ее, вы узнаете, как планировать проект, как оценить возможные риски и внести свой вклад в успешную работу всего коллектива. Автору книги, Мюррею Кентору, удалось сделать ее достаточно универсальной. Благодаря этому книга может стать настольной не только для менеджеров программных продуктов, но и для коллективов разработчиков, а также для всех тех, кого пугает обилие технического материала, связанного с процессом разработки. В книге представлен материал, достаточный для понимания процесса в целом и для подготовки грамотных менеджеров, способствующих успешной реализации проектов. Реализацией какого проекта вы бы ни занимались, в этой книге вы получите ценное практическое руководство для своей работы и благодаря ей станете настоящим лидером в коллективе.

Антон Элиенс



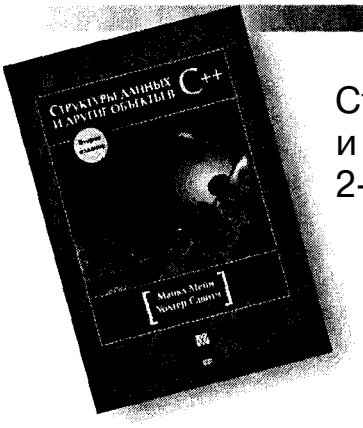
Принципы объектно-ориентированной разработки программ, 2-е издание

В продаже

В последние годы технология объектно-ориентированного программирования (ООП) заняла лидирующие позиции в области разработки программного обеспечения. Эта книга содержит исчерпывающее описание как преимуществ, так и недостатков объектно-ориентированного подхода, последовательно рассматриваемых в отношении отдельных стадий жизненного цикла программного обеспечения, начиная от анализа требований и заканчивая сопровождением и модернизацией готовых программ. В каждом случае автор стремится увязать соответствующие принципы ООП с существующей практикой прикладного программирования.

Предлагаемый материал богато иллюстрирован примерами на языках Java и C++, и включает обсуждение основных концепций таких объектно-ориентированных языков, как Smalltalk, Eiffel, C++, Java, а также UML и технологии CORBA.
Для лучшего закрепления материала в книгу включены подборки вопросов для самопроверки, а также практические примеры разработки мультимедиа- и Web-приложений. Книга может быть полезна как студентам, изучающим соответствующие курсы, так и специалистам-практикам в области разработки программного обеспечения.

Майкл Мейн, Уолтер Савитч



Структуры данных и другие объекты в C++, 2-е издание

Плановая дата выхода — 3-й квартал 2002 г.

Этот курс приобрел большую популярность, о чем свидетельствует выход его второго издания. Авторам удалось найти удачный баланс между стремлением к исчерпывающему предоставлению материала о структурах данных и объектно-ориентированном программировании с одной стороны, и максимально доступным стилем изложением с другой.

Данная книга является продолжением базового курса изучения языка C++, написанного тем же автором. Основной упор в ней делается на спецификации, проектировании, реализации и использовании основных типов данных языка C++. Кроме того, описывается широкий диапазон объектно-ориентированных технологий программирования, методов сортировки и анализа алгоритмов. Предполагается, что читатель уже имеет определенный уровень знаний как по основам компьютерных наук, так и по самому языку C++.

Книга будет полезна широкому кругу читателей, как студентов, так профессиональных программистов.