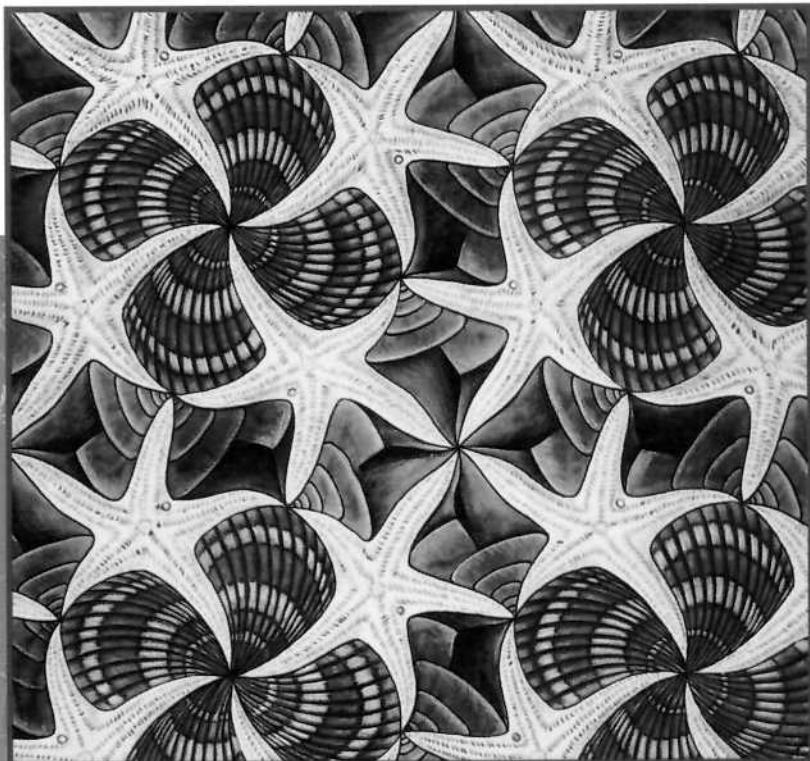


ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Новый подход к объектно-ориентированному
анализу и проектированию



Алан Шаллоуей
Джеймс Р. Тротт

СЕРИЯ "ШАБЛОНЫ ПРОЕКТИРОВАНИЯ"

Design Patterns Explained

A New Perspective on Object-Oriented Design

ALAN SHALLOWAY
JAMES R. TROTT



ADDISON-WESLEY PUBLISHING COMPANY

*Boston • San Francisco • New York
London • Toronto • Sydney • Tokyo • Singapore • Madrid
Mexico City • Munich • Paris • Cape Town • Hong Kong • Montreal*

32.973
Ш 18

Шаблоны проектирования

*Новый подход к объектно-ориентированному
анализу и проектированию*

38956

АЛАН ШАЛЛОУЕЙ
ДЖЕЙМС Р. ТРОТТ



038956

Національний банк України
Центральна бібліотека



Москва • Санкт-Петербург • Киев
2002

ББК 32.973.26-18.2.75

Ш18

УДК 681.3.07

Издательский дом "Вильямс"

Перевод с английского С.А. Мельника, К.В. Сальникова, А.В. Слепцова

Под редакцией А.В. Слепцова

По общим вопросам обращайтесь в Издательский дом "Вильямс"

по адресу: info@williamspublishing.com,

<http://www.williamspublishing.com>

Шаллоуей, Аллан, Тротт, Джеймс, Р.

Ш18 Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию: Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 288 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0301-7 (рус.)

Объектно-ориентированное программирование с использованием шаблонов проектирования призвано облегчить работу проектировщиков и разработчиков программного обеспечения. Но изучение и успешное использование этих методов может оказаться достаточно сложным делом. Эта книга дает точное представление о десяти наиболее важных шаблонах проектирования, которые никогда не используются самостоятельно, а только во взаимодействии друг с другом, что и гарантирует надежность создаваемых приложений. Полученных знаний будет вполне достаточно для дальнейшего изучения литературы по шаблонам проектирования и даже для создания своих собственных шаблонов.

Книга предназначена как для профессиональных разработчиков ПО, так и для студентов, изучающих основы ООП.

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 Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Pearson Education Limited, Copyright © 2002

All rights reserved. No part of this book may be reproduced, stored in retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without either the prior written permission or the Publisher.

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

ISBN 5-8459-0301-7 (рус.)

ISBN 0-2017-1594-5 (англ.)

© Издательский дом "Вильямс", 2002

© Addison-Wesley, 2002

Ограниченност традиционного объектно- ориентированного проектирования

В этой части

В этой части мы обсудим решение некоторых задач, взятых из реальной жизни, с использованием стандартных методов объектно-ориентированной разработки. Над обсуждаемой ниже проблемой я работал в то время, когда только приступал к изучению шаблонов проектирования.

| Глава | Предмет обсуждения |
|-------|---|
| 3 | <ul style="list-style-type: none">• Описание характерной для систем автоматизированного проектирования (САПР) проблемы: выборка информации из больших производственных баз данных систем САПР для нужд сложных и дорогих прикладных программ анализа• Поскольку программное обеспечение систем САПР постоянно развивается и модернизируется, упомянутая выше проблема неразрешима без создания гибкого (настраиваемого) кода |
| 4 | <ul style="list-style-type: none">• Первое решение связанной с системой САПР проблемы, выполненное с использованием стандартных объектно-ориентированных методов• Работая над указанной проблемой, я еще не полностью разобрался с теми принципами, которые были положены в основу многих шаблонов проек- |

Глава 5

Первое знакомство с шаблонами проектирования

Введение

В этой главе мы познакомимся с концепцией шаблонов проектирования.

Здесь будут рассмотрены:

- концепция шаблонов проектирования, их первоначальное появление в архитектуре и последующее распространение на область проектирования программного обеспечения;
- мотивы для изучения шаблонов проектирования.

Шаблоны проектирования – это один из важнейших компонентов объектно-ориентированной технологии разработки программного обеспечения. Они широко применяются в инструментах анализа, подробно описываются в книгах и часто обсуждаются на семинарах по объектно-ориентированному проектированию. Существует множество групп и курсов подготовки по их изучению. Часто приходится слышать, что приступить к изучению шаблонов проектирования следует только после того, как будут приобретены определенные навыки применения объектно-ориентированной технологии. Однако, по мнению автора, истинно обратное утверждение: предварительное изучение шаблонов помогает лучше понять объектно-ориентированное проектирование и анализ.

В остальной части книги мы будем обсуждать не только шаблоны проектирования, но и то, как в них проявляются и утверждаются принципы объектно-ориентированной технологии. Эта книга с одной стороны должна помочь читателю глубже понять эти принципы, а с другой – проиллюстрировать, каким образом обсуждаемые в ней шаблоны проектирования могут способствовать созданию хороших проектов.

Предлагаемый ниже материал в некоторых случаях может показаться чересчур абстрактным или философским. Но читателю следует набраться терпения, поскольку в этой главе обсуждаются те основы, которые просто необходимы для понимания шаблонов проектирования. Кроме того, изучение этого материала поможет вам лучше понять принципы построения и методы работы с новыми шаблонами.

Многие из идей, изложенных здесь, были найдены автором в книге Кристофера Александера (Christopher Alexander) *The Timeless Way of Building*¹. Эти идеи будут обсуждаться в различных главах книги.

¹ Alexander C., Ishikawa S., Silverstein M. *The Timeless Way of Building*, New York, NY: Oxford University Press, 1979.

Шаблоны проектирования пришли из области архитектуры и культурологии

Много лет назад архитектор по имени Кристофер Александр задумался над вопросом: "Является ли качество объективной категорией?". Следует ли считать представление о красоте сугубо индивидуальным, или люди могут прийти к общему соглашению, согласно которому некоторые вещи будут считаться красивыми, а другие нет? Александр размышлял о красоте с точки зрения архитектуры. Его интересовало, по каким показателям мы оцениваем архитектурные проекты. Например, если некто вознамерился спроектировать крыльцо дома, то как он может получить гарантии, что созданный им проект будет хорош? Можем ли мы знать заранее, что проект будет действительно хорош? Имеются ли объективные основания для вынесения такого суждения? Существует ли необходимая основа для достижения общего согласия?

Александр принял как постулат, что в области архитектуры такое объективное основание существует. Суждение о том, что некоторое здание является красивым, – это не просто вопрос вкуса. Красоту можно описать с помощью объективных критериев, которые могут быть измерены.

К похожим выводам пришли и исследователи в области в культурологии. В пределах одной культуры большинство индивидуумов имеют схожие представления о том, что является сделанным хорошо и что является красивым. В основе их суждений есть нечто более общее, чем сугубо индивидуальные представления о красоте. Похоже, что существуют некоторые трансцендентные образы или шаблоны, являющиеся объективным основанием для оценки предметов. Основная задача культурологии состоит в описании подобных шаблонов, определяющих каноны поведения и систему ценности каждой из культур.²

Исходная посылка создания шаблонов проектирования также состояла в необходимости объективной оценки качества программного обеспечения.

Если идея о том, что оценить и описать высококачественный проект возможно, возражений не вызывает, то не пора ли попробовать создать нечто подобное? Я полагаю, Александр сформулировал для себя следующие вопросы.

1. *Что есть такого в проекте хорошего качества, что отличает его от плохого проекта?*
2. *Что именно отличает проект низкого качества от проекта высокого качества?*

Эти вопросы навели Александера на мысль о том, что если качество проекта является объективной категорией, то мы можем явно определить, что именно делает проекты хорошими, а что – плохими.

Александр изучал эту проблему, обследуя множество зданий, городов, улиц и всего прочего, что люди построили для своего проживания. В результате он обнаружил, что все, что было построено хорошо, имело между собой нечто общее.

Архитектурные структуры отличаются друг от друга, даже если они относятся к одному и тому же типу. Однако, не взирая на имеющиеся различия, они могут оставаться высококачественными.

² Антрополог Рут Бенедикт (*Ruth Benedict*) была одним из создателей метода анализа культур с помощью выявления существующих в ней шаблонов. Например, см. Benedict R., *The Chrysanthemum and the Sword*, Boston, MA: Houghton Mifflin, 1946.

Например, два крыльца могут выглядеть конструктивно различными и, тем не менее, обладать высоким качеством – просто они решают *различные задачи* в различных зданиях. Одно крыльцо может служить для прохода с тротуара ко входной двери, а назначение другого может состоять в создании тени жарким днем. В другом случае два крыльца могут решать одну и ту же задачу, но *различными способами*.

Александр понял это и пришел к выводу, что сооружения нельзя рассматривать обособленно от проблемы, для решения которой они предназначены. Поэтому в своих попытках найти и описать критерии красоты и качества проекта он стал рассматривать различные архитектурные элементы, предназначенные для решения одинаковых задач. Например, на рис. 5.1 продемонстрированы два различных варианта оформления входа в здание.

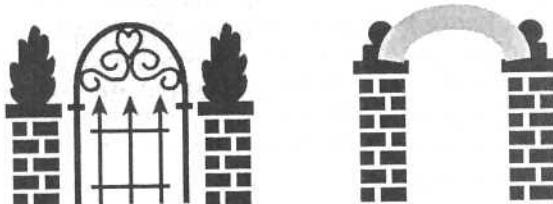


РИС. 5.1. Архитектурные элементы могут выглядеть различными, но выполнять одну функцию

Александр установил, что, сфокусировав внимание на структурах, предназначенных для решения подобных задач, можно обнаружить сходство между различными проектами, которым присуще высокое качество. Он назвал эти сходства *шаблонами*.

Он определил понятие шаблона как "решение проблемы в контексте".³

Каждый шаблон описывает проблему, которая возникает в данной среде снова и снова, а затем предлагает принцип ее решения таким способом, который можно будет применять многократно, получая каждый раз результаты, не повторяющие друг друга.

В табл. 5.1 приведены размышления Александера, взятые мной из его работы⁴. Они иллюстрируют сделанное выше утверждение.

Таблица 5.1. Выдержка из книги "Строительство на века"

| Слова Александера | Мой комментарий |
|---|--|
| Аналогичным образом, правильно спроектированный внутренний двор дома способствует отдыху и восстановлению сил его жителей | Шаблон всегда имеет уникальное имя и предназначен для определенной цели. В данном случае шаблон называется "Внутренний двор", а его назначение состоит в предоставлении жителям дома места для отдыха и восстановления сил |

³ Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Building, New York, NY: Oxford University Press, 1979.

⁴ Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Building, New York, NY: Oxford University Press, 1979.

Продолжение таблицы

| Слова Александера | Мой комментарий |
|---|--|
| <p>Рассмотрим, чем занимаются люди во внутреннем дворе. Прежде всего, они ищут возможность уединиться на свежем воздухе и найти такое место, где можно будет посидеть под открытым небом, полюбоваться звездами, насладиться солнечным теплом или посадить цветы. Это совершенно очевидно</p> | <p>Хотя иногда это может быть вполне очевидно, нам важно четко сформулировать ту основную задачу, которая может быть решена с помощью данного шаблона. Именно это делает Александр при обсуждении шаблона "Внутренний Двор"</p> |
| <p>Но не следует забывать и о других, на первый взгляд, менее важных задачах. В случае, если двор слишком замкнут и не дает необходимого обзора, люди в нем чувствуют себя неуютно. У них возникает желание покинуть это место, поскольку они нуждаются в более широком поле обзора</p> | <p>Здесь подчеркиваются трудности, связанные с упрощенным подходом к решению задачи, а затем предлагается лучший способ решения, позволяющий избежать указанных затруднений</p> |
| <p>Однако привычка — вторая натура. Когда люди в своей обыденной жизни изо дня в день проходят через внутренний двор множество раз, это место становится для них знакомым, т.е. привычным местом прохода — и именно для этих целей оно будет использоваться ими в дальнейшем</p> | <p>Привычка очень часто мешает нам увидеть очевидные вещи. Ценность шаблонов в том, что люди, не обладающие большим опытом, могут воспользоваться преимуществами, найденными их более опытными коллегами. Шаблоны позволяют им определять, какие детали должны быть включены для достижения высокого качества проекта, а также чего следует избегать, чтобы не потерять это качество</p> |
| <p>Внутренний двор, не имеющий выхода на улицу, становится местом, которое посещают только изредка, когда хочется именно туда. Он остается непривычным местом, которое используется все реже и реже, поскольку люди, в основном, склонны к посещению знакомых им мест.</p> | |
| <p>Кроме того, есть что-то неприятное, нежелательное в том, чтобы из дома выйти сразу на улицу. Это вроде бы мелочь, но ее достаточно, чтобы вызвать нервозность или раздражение.</p> | |
| <p>Если есть дополнительное пространство в виде крыльца или веранды под навесом, но открытое для воздуха — с психологической точки зрения это промежуточное состояние между домом и улицей. В результате человеку становится легче и проще сделать тот короткий шаг, которые выведет его во внутренний двор</p> | <p>Предлагается решение, способное изменить ваше мнение о преимуществах хорошего внутреннего двора</p> |

Окончание таблицы

| Слова Александера | Мой комментарий |
|--|--|
| <p>Если внутренний двор позволяет взглянуть на окружающее пространство, представляет собой удобный путь между различными комнатами, включает в себя некоторый вариант крыльца или веранды, то это наполняет его существование смыслом. Внешний обзор делает посещение внутреннего двора приятным, а пересечение в нем многих путей между помещениями обеспечивает этим посещениям чувство привычности. Наличие калитки или веранды придает дополнительные психологические удобства при выходе из дома на улицу</p> | <p>Александр рассказывает, как построить отличный внутренний двор...</p> <p>...и поясняет, почему он будет так хорош</p> |

В качестве заключения укажем четыре компонента, которые, по мнению Александра, должны присутствовать в описании каждого шаблона.

- Имя шаблона.
- Назначение шаблона и описание задачи которую он призван решать.
- Способ решения поставленной задачи.
- Ограничения и требования, которые необходимо принимать во внимание при решении задачи.

Александр принял как постулат, что с помощью шаблонов может быть решена любая архитектурная задача, с которой столкнется проектировщик. Затем он пошел дальше и высказал утверждение о том, что совместное использование нескольких шаблонов позволит решать комплексные архитектурные проблемы.

Как можно применить одновременно несколько шаблонов, мы обсудим на страницах этой книги позднее. Сейчас же сосредоточимся на пользе отдельных шаблонов при решении специализированных проблем.

Переход от архитектурных шаблонов к шаблонам проектирования программного обеспечения

Но какое отношение может иметь весь этот архитектурный материал к специалистам в области программного обеспечения, коими мы являемся?

В начале 1990-х некоторые из опытных разработчиков программного обеспечения ознакомились с упоминавшейся выше работой Александера об архитектурных шаблонах. Они задались вопросом, возможно ли применение идеи архитектурных шаблонов при реализации проектов в области создания программного обеспечения.⁵

⁵ Консорциум ESPRIT в Европе обратился к этому же вопросу еще в 1980-х. Проекты ESPRIT с номерами 1098 и 5248 состояли в разработке методологии проектирования с использованием шаблонов,

Сформулируем те вопросы, на которые требовалось получить ответ.

- Существуют ли в области программного обеспечения проблемы, возникающие снова и снова, и могут ли они быть решены тем же способом?
- Возможно ли проектирование программного обеспечения в терминах шаблонов — т.е. создание конкретных решений на основе тех шаблонов, которые будут выявлены в поставленных задачах?

Интуиция подсказывала исследователям, что ответы на оба эти вопроса определенно будут положительными. Следующим шагом необходимо было идентифицировать несколько подобных шаблонов и разработать стандартные методы каталогизации новых.

Хотя в начале 1990-х над шаблонами проектирования работали многие исследователи, наибольшее влияние на это сообщество оказала книга Гаммы, Хелма, Джонсона и Влиссайдеса *Шаблоны проектирования: элементы многократного использования кода в объектно-ориентированном программировании*.⁶ Эта работа приобрела очень широкую известность. Свидетельством ее популярности служит тот факт, что четыре автора книги получили шутливое прозвище "банда четырех".

В книге рассматривается несколько важных аспектов проблемы.

- Применение идеи шаблонов проектирования в области разработки программного обеспечения.
- Описание структур, предназначенных для каталогизации и описания шаблонов проектирования.
- Обсуждение 23 конкретных шаблонов проектирования.
- Формулирование концепций объектно-ориентированных стратегий и подходов, построенных на применении шаблонов проектирования.

Важно понять, что авторы сами не создавали тех шаблонов, которые описаны в их книге. Скорее, они идентифицировали эти шаблоны как уже существующие в разработках, выполненных сообществом создателей программного обеспечения. Каждый из шаблонов отражает те результаты, которые были достигнуты в высококачественных проектах при решении определенных, специфических проблем (этот подход напоминает нам о работе Александера).

На сегодня существует несколько различных форм описания шаблонов проектирования. Поскольку эта книга вовсе не о том, как следует описывать шаблоны проектирования, мы не приводим здесь нашего мнения по поводу того, какая из структур описания шаблонов является оптимальной. Однако пункты, приведенные в табл. 5.2, безусловно, должны присутствовать в любом описании шаблона проектирования.

получившей название KADS (*Knowledge Analysis and Support* — анализ и поддержка знаний). Эта методология предусматривала использование шаблонов при создании экспертизных систем. Карен Гарднер (*Karen Gardner*) распространила аналитические шаблоны KADS на объектную технологию. См. *Gardner K. Cognitive Patterns: Problem Solving Frameworks for Object Technology*, New York, NY: Cambridge University Press, 1998.

⁶ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995.

Каждый шаблон, упоминаемый в этой книге, будет представлен в виде краткого резюме, описывающего его важнейшие характеристики.

Таблица 5.2. Важнейшие характеристики шаблонов

| Характеристика | Описание |
|----------------|--|
| Имя | Все шаблоны имеют уникальное имя, служащее для их идентификации |
| Назначение | Назначение данного шаблона |
| Задача | Задача, которую шаблон позволяет решить |
| Способ решения | Способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден |
| Участники | Сущности, принимающие участие в решении задачи |
| Следствия | Последствия от использования шаблона как результат действий, выполняемых в шаблоне |
| Реализация | Возможный вариант реализации шаблона. <i>Замечание.</i> Реализация является лишь конкретным воплощением общей идеи шаблона и не должна пониматься как собственно сам шаблон |
| Ссылки | Место в книге "банды четырех", где можно найти дополнительную информацию |

Следствия и действия

Следствия — это термин, который широко используется в отношении шаблонов проектирования, но часто неправильно истолковывается. В применении к шаблонам проектирования под следствием понимается любой эффект, вызванный какой-либо причиной в процессе его функционирования. Иначе говоря, если шаблон реализован так-то и так-то, то следствиями его использования будут результаты любых выполняемых в нем действий.

Зачем нужно изучать шаблоны проектирования

Теперь, когда мы знаем, что такое шаблоны проектирования, можно попытаться ответить на вопрос, зачем нужно их изучать. На то имеется несколько причин, часть из которых вполне очевидна, тогда как об остальных этого не скажешь.

Чаще всего причины, по которым следует изучать шаблоны проектирования, формулируют следующим образом.

- *Возможность многократного использования.* Повторное использование решений из уже завершенных успешных проектов позволяет быстро приступить к решению новых проблем и избежать типичных ошибок. Разработчик получает прямую выгоду от использования опыта других разработчиков, избежав необходимости вновь и вновь изобретать велосипед.

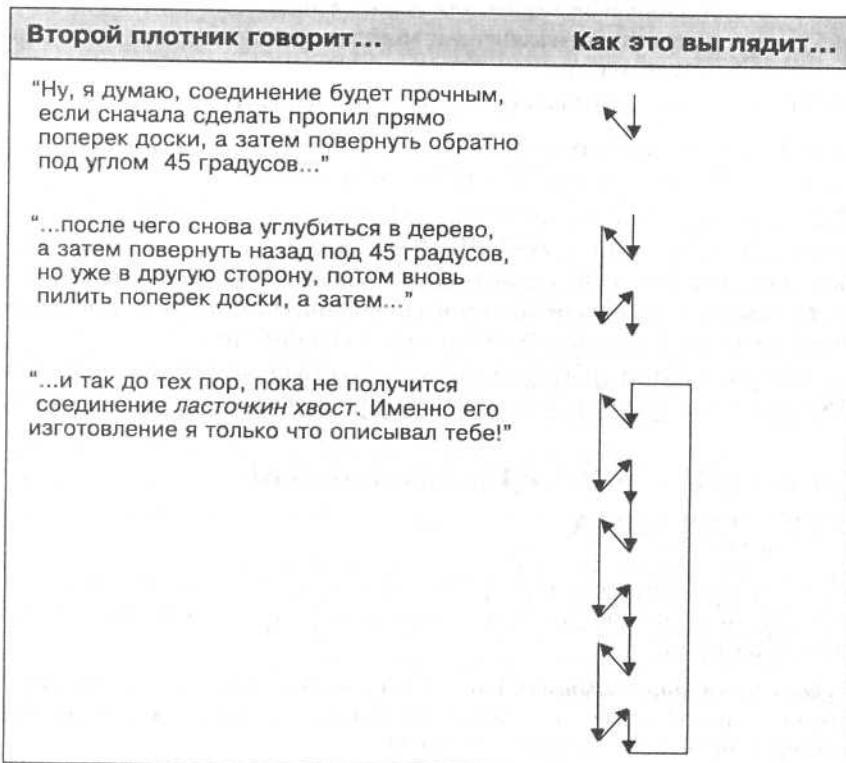
- *Применение единой терминологии.* Профессиональное общение и работа в группе (команде разработчиков) требует наличия единого базового словаря и единой точки зрения на проблему. Шаблоны проектирования предоставляют подобную общую точку зрения как на этапе анализа, так и при реализации проекта.

Однако существует и третья причина, по которой следует изучать шаблоны проектирования.

Шаблоны проектирования предоставляют нам абстрактный высокоуровневый взгляд как на проблему, так и на весь процесс объектно-ориентированной разработки. Это помогает избежать излишней детализации на ранних стадиях проектирования.

Я надеюсь, что, прочитав эту книгу, вы согласитесь с тем, что именно последняя причина является важнейшей для изучения шаблонов проектирования. Я постараюсь изменить ваш образ мышления за счет усиления и развития у вас навыков системного аналитика.

Для иллюстрации сказанного выше представим себе разговор двух плотников о том, как сделать выдвижные ящики для письменного стола.⁷



⁷ Этот диалог взят автором из рассказа Ральфа Джонсона (Ralph Johnson) и соответствующим образом переработан.

Плотник 1. Как, по твоему мнению, следует сделать эти ящики?

Плотник 2. Ну, я думаю, соединение будет прочным, если сначала сделать пропил прямо поперек доски, а затем повернуть обратно под углом 45 градусов, после чего снова углубиться в дерево, а затем повернуть назад под углом 45 градусов, но уже в другую сторону, потом вновь пилить поперек доски, а затем...

Попробуйте понять, о чём здесь идет речь!

Не правда ли, довольно запутанное описание? Что именно второй плотник имеет в виду? Чрезмерное количество подробностей мешает уловить смысл обсуждаемого. Попытаемся представить это описание графически.

Не напоминает ли это вам устное описание фрагмента программного кода? Вероятно, программист сказал бы что-то, похожее на следующее:

...а затем я использую цикл WHILE, чтобы выполнить..., за которым следует несколько операторов IF, предназначенных для..., а потом я использую оператор SWITCH для обработки...

Здесь представлено подробное описание фрагмента программы, но из него остается абсолютно неясным, что эта программа делает и для чего!

Конечно, никакой уважающий себя плотник не стал бы говорить ничего подобного тому, что было приведено выше.

В действительности мы услышали бы примерно такой диалог.

Плотник 1. Какой вариант соединения мы будем использовать, — "ласточкин хвост" или простое угловое соединение под 45 градусов?

Нетрудно заметить, что новый вариант качественно отличается от прежнего. Плотники обсуждают отличия в качестве решения проблемы, поэтому в данном случае их общение проходит на более высоком, а значит, и более абстрактном уровне. Они не тратят время и силы на обсуждение специфических деталей отдельных типов соединений и не вдаются в излишние подробности их описания.

Когда плотник говорит о простом соединении деревянных деталей под углом 45 градусов, он понимает, что такому решению свойственны следующие характеристики.

- *Это более простое соединение.* Угловое соединение является более простым в изготовлении. Достаточно обрезать торцы соединяемых деталей под углом в 45 градусов, состыковать их, а затем скрепить между собой с помощью гвоздей или клея — как показано на рис. 5.2.
- *Это соединение имеет меньшую прочность.* Угловое соединение является менее прочным, чем соединение "ласточкин хвост", и не способно выдерживать большую нагрузку.
- *Это более незаметное соединение.* Единственный стык в месте соединения деталей меньше заметен глазу, чем многочисленные распилы в соединении типа "ласточкин хвост".

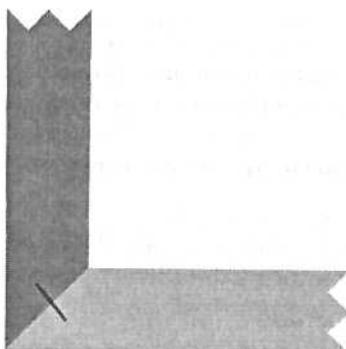


РИС. 5.2. Простое соединение под углом 45 градусов

Когда плотник говорит о соединении типа "ласточкин хвост" (способ выполнения которого был описан выше), он представляет себе его характеристики, которые могут быть вовсе не очевидны для обывателя, но, безусловно, известны любому другому плотнику.

- *Это более сложный вариант.* Выполнить это соединение сложнее, чем простое угловое, поэтому оно дороже.
- *Это соединение устойчиво по отношению к температуре и влажности.* При изменении температуры и влажности воздуха древесина расширяется или сжимается, однако это не оказывает влияния на прочность соединения данного типа.
- *Это соединение не нуждается в дополнительном укреплении.* Благодаря точному соответствию вырезов скрепляемых деталей по отношению друг к другу соединение фактически не нуждается в дополнительном укреплении kleem или гвоздями.
- *Это соединение более эстетично.* Если соединение этого типа выполнено качественно, оно оставляет очень приятное впечатление.

Другими словами, соединение типа "ласточкин хвост" надежно и красиво, но менее технологично, а поэтому изготовление его обходится дороже.

Таким образом, когда первый плотник задает вопрос:

Какой способ соединения мы будем использовать – типа "ласточкин хвост" или обычное, под углом в 45 градусов?

В действительности он спрашивает:

Мы воспользуемся более сложным и дорогим, но красивым и долговечным соединением или сделаем все быстро, не заботясь о высоком качестве, лишь бы ящик продержался до тех пор, пока нам не заплатят?

Можно сказать, что в действительности общение плотников происходит на двух уровнях: внешнем уровне произносимых слов и более высоком уровне (мета-уровне) реальной беседы. Последний скрыт от обывателя и намного богаче первого по содержанию. Этот более высокий уровень – назовем его уровнем "шаблонов плотников" – отражает реальные вопросы разработки проекта в нашем примере с плотниками.

В первом варианте второй плотник затеняет реально обсуждаемые проблемы, углубляясь в детали выполнения соединения. Во втором варианте первый плотник предлагает принять решение о выборе типа соединения, исходя из его стоимости и качества.

Чей подход более эффективен? С кем, по вашему мнению, предпочтительнее работать?

Шаблоны позволяют нам видеть лес за деревьями, поскольку помогают поднять уровень мышления. Ниже в этой книге будет показано, что, когда удается подняться на более высокий уровень мышления, разработчику становятся доступны новые методы проектирования. Именно в этом состоит реальная сила шаблонов проектирования.

Другие преимущества применения шаблонов проектирования

Мой опыт работы в группах разработчиков показал, что в результате применения шаблонов проектирования повышается эффективность труда отдельных исполнителей и всей группы в целом. Это происходит из-за того, что начинающие члены группы видят на примере более опытных разработчиков, как шаблоны проектирования могут применяться и какую пользу они приносят. Совместная работа дает новичкам стимул и реальную возможность быстрее изучить и освоить эти новые концепции.

Применение многих шаблонов проектирования позволяет также создавать более модифицируемое и гибкое программное обеспечение. Причина состоит в том, что эти решения уже испытаны временем. Поэтому использование шаблонов позволяет создавать структуры, допускающие их модификацию в большей степени, чем это возможно в случае решения, первым пришедшего на ум.

Шаблоны проектирования, изученные должным образом, существенно помогают общему пониманию основных принципов объектно-ориентированного проектирования. Я убеждался в этом неоднократно в процессе преподавания курса объектно-ориентированного проектирования. Курс я начинал с краткого представления объектно-ориентированной парадигмы, а затем переходил к освещению темы шаблонов проектирования, используя основные концепции ООП (инкапсуляция, наследование и полиморфизм) лишь для иллюстрации излагаемого материала. К концу трехдневного курса, хотя речь на лекциях шла главным образом о шаблонах, базовые концепции ООП также становились понятными и знакомыми всем слушателям.

"Бандой четырех" было предложено несколько стратегий создания хорошего объектно-ориентированного проекта. В частности, они предложили следующее.

- Проектирование согласно интерфейсам.
- Предпочтение синтеза наследованию.
- Выявление изменяющихся величин и их инкапсуляция.

Эти стратегии использовались в большинстве шаблонов проектирования, обсуждаемых в данной книге. Для оценки полезности указанных стратегий вовсе не обязательно изучить большое количество шаблонов — достаточно всего нескольких. Приобретенный опыт позволит применять новые концепции и к задачам собственных проектов, даже без непосредственного использования шаблонов проектирования.

Еще одно преимущество состоит в том, что шаблоны проектирования позволяют разработчику или группе разработчиков находить проектные решения для сложных проблем, не создавая громоздкой иерархии наследования классов. Даже если шаблоны не используются в проекте непосредственно, одно только уменьшение размера иерархий наследования классов уже будет способствовать повышению качества проекта.

Резюме

В этой главе мы выяснили, что представляют собой шаблоны проектирования. Кристофер Александр сказал: "Шаблоны – это решение проблемы в контексте". Шаблон проектирования это нечто большее, чем просто способ решения какой-либо проблемы. Шаблон проектирования – это способ представления в общем виде как условия задачи, которую необходимо решить, так и правильных подходов к ее решению.

Мы также рассмотрели причины для изучения шаблонов проектирования. Шаблоны могут помочь разработчику в следующем.

- Многократно применять высококачественное решение для повторяющихся задач.
- Ввести общую терминологию для расширения взаимопонимания в пределах группы разработчиков.
- Поднять уровень, на котором проблема решается, и избежать нежелательного углубления в детали реализации уже на ранних этапах разработки.
- Оценить, что было создано – именно то, что нужно, или же просто некоторое работоспособное решение.
- Ускорить профессиональное развитие как всей группы разработчиков в целом, так и отдельных ее членов.
- Повысить модифицируемость кода.
- Обеспечить выбор лучших вариантов реализации проекта, даже если сами шаблоны проектирования в нем не используются.
- Найти альтернативное решение для исключения громоздких иерархий наследования классов.

Глава 8

Расширение горизонтов

Введение

В предыдущих главах уже упоминалось о трех фундаментальных концепциях объектно-ориентированного проектирования – объектах, инкапсуляции и абстрактных классах. Очень важно, как проектировщик применяет эти концепции. Традиционный подход к пониманию указанных терминов ограничивает возможности разработки. В этой главе мы вернемся на шаг назад и более подробно рассмотрим темы, уже обсуждавшиеся ранее. Здесь так же раскрываются особенности нового взгляда на объектно-ориентированное проектирование, связанные с появлением шаблонов проектирования.

В этой главе мы выполним следующее.

- Сравним в противопоставлении следующие подходы:
 - традиционный, рассматривающий объекты как совокупность данных и методов, и новый, понимающий под объектом предмет, имеющий некоторые обязательства;
 - традиционный, рассматривающий инкапсуляцию исключительно как механизм сокрытия данных, и новый, понимающий под инкапсуляцией способность к сокрытию чего угодно. Особенно важно понимать, что инкапсуляция может быть применена и для сокрытия различий в поведении;
 - традиционный способ применения механизма наследования с целью реализации специализации и повторного использования кода и новый, понимающий под наследованием метод классификации объектов.
- Проанализируем, как новый подход позволяет объектам демонстрировать различное поведение.
- Рассмотрим, как различные уровни проектирования – концептуальный, спецификаций и реализации – соотносятся с абстрактным классом и его производными классами.

Вероятно, предложенный здесь новый подход вовсе не является оригинальным. Я уверен, что именно он позволил создателям шаблонов проектирования разработать ту концепцию, которую сейчас принято называть шаблоном. Несомненно, что этот подход был известен и Кристоферу Александеру, и Джиму Коплину (Jim Coplien), и "банде четырех".

Хотя предлагаемый здесь подход нельзя считать оригинальным, я уверен, что он никогда не обсуждался ранее в таком виде, как это сделано в данной книге. В отличие

от других авторов я предпринял попытку отделить рассмотрение шаблонов от обсуждения особенностей их поведения.

Говоря о новом подходе, я подразумеваю, что для большинства разработчиков это, вероятно, совершенно новый взгляд на объектно-ориентированное проектирование. По крайней мере, для меня он был таковым, когда я впервые приступил к изучению шаблонов проектирования.

Объекты: традиционное представление и новый подход

Традиционное представление об объектах состоит в том, что они представляют собой совокупность данных и методов их обработки. Один из моих преподавателей назвал их "умными данными". Лишь один шаг отделяет их от баз данных. Подобное представление возникает при взгляде на объекты с точки зрения их реализации.

Хотя данное определение является совершенно точным – как объяснялось выше, в главе 1, *Объектно-ориентированная парадигма*, – оно построено при взгляде на объекты с точки зрения их реализации. Более полезным является определение, построенное при взгляде на объекты с концептуальной точки зрения, когда объект рассматривается как сущность, имеющая некоторые обязательства. Эти обязательства определяют поведение объекта. В некоторых случаях мы также будем представлять объект как сущность, обладающую конкретным поведением.

Такое определение предпочтительнее, поскольку оно помогает сосредоточиться на том, что объект должен *делать*, не задаваясь преждевременно вопросом о том, как это можно реализовать. Подобный подход позволяет разделить процедуру разработки программного обеспечения на два этапа.

1. Создание предварительного проекта без излишней детализации всех процессов.
2. Реализация разработанного проекта в кодах.

В конечном счете, этот подход позволяет более точно выбрать и определить объект (в смысле отправной точки любого проекта). Второе определение объекта является более гибким, поскольку позволяет сосредоточиться на том, что объект делает, а механизм наследования предоставляет инструмент реализации его поведения по мере необходимости. При взгляде на объект с точки зрения реализации также можно достичь этого, но гибкость в этом случае, как правило, обеспечивается более сложным путем.

Гораздо проще рассуждать в терминах обязательств, так как это помогает определить открытый интерфейс объекта. Если объект обладает обязательствами, то очевидно, что должен существовать какой-то способ попросить его выполнить данные обязательства. Однако это требование никак не соотносится с тем, что находится внутри объекта. Информация, с которой связаны обязательства объекта, вовсе не обязательно должна находиться непосредственно в самом этом объекте.

Предположим, что существует объект **Shape**, имеющий такие обязательства.

- Знать свое местоположение.
- Уметь отобразить себя на экране монитора.
- Уметь удалить свое изображение с экрана монитора.

Этот набор обязательств предполагает существование определенного набора методов, необходимых для их реализации:

- метод `GetLocation(...);`
- метод `DrawShape(...);`
- метод `UnDrawShape(...).`

Для нас не имеет никакого значения, что именно находится внутри объекта `Shape`. Единственное, что нас интересует, это чтобы объект `Shape` обеспечивал требуемое поведение. Для этой цели могут использоваться атрибуты внутри объекта или методы, вычисляющие требуемые результаты или даже обращающиеся к другим объектам. Таким образом, объект `Shape` может как содержать атрибуты, описывающие его месторасположение, так и обращаться к другим объектам базы данных, чтобы получить сведения о своем местоположении. Такой подход предоставляет высокую гибкость, необходимую для эффективного моделирования процессов в проблемной области.

Интересно отметить, что перенос акцента с реализации на мотивацию – характерная черта описания шаблонов проектирования.

Стремитесь всегда рассматривать объекты в указанном ракурсе. Выбор такого подхода в качестве основного позволит вам создавать превосходные проекты.

Инкапсуляция: традиционное представление и новый подход

На лекциях по курсу проектирования с применением шаблонов я часто обращаюсь к студентам со следующим вопросом: "Кто из вас знаком с определением инкапсуляции как механизма сокрытия данных?". В ответ почти каждый человек в аудитории поднимает руку.

Затем я рассказываю им историю о моем зонтике. Следует заметить, что живу я в городе Сиэтл, штат Вашингтон, который отличается очень влажным климатом. Поэтому зонтики и плащи с капюшоном – вещь, совершенно необходимая жителям этого города осенью, зимой и весной.

Так вот, мой зонтик довольно большой – фактически, вместе со мной под ним могут найти укрытие еще три или четыре человека. Спрятавшись от дождя под этим зонтиком, мы можем перемещаться из одного помещения в другое, оставаясь сухими. Зонтик оборудован акустической стереосистемой, помогающей не скучать, пока мы находимся под его защитой. Представьте себе, что в нем имеется даже система кондиционирования воздуха, регулирующая окружающую температуру. Одним словом, это весьма комфорtabельный зонтик.

Зонтик чрезвычайно удобен и всегда ждет меня там, где я его оставляю. У него есть колесики и его не нужно переносить с места на место в руках. Более того, перемещение этого зонтика происходит вообще без моего участия, поскольку он оборудован собственным двигателем. Если нужно, можно даже открыть специальный люк вверху моего зонтика, чтобы впустить под него лучи солнца. (Почему я пользуюсь зонтиком при солнечной погоде, я вам объяснять не стану.)

В Сиэтле есть сотни тысяч таких зонтиков различных типов и цветов.

Большинство людей называет их *автомобилями*.

Однако лично я думаю о своем автомобиле как о зонтике, ведь зонтик – это предмет, которым пользуются, чтобы укрыться от дождя. Каждый раз, когда я ожидаю кого-нибудь на улице во время дождя, я сижу в моем "зонтике", чтобы оставаться сухим!

Конечно, в действительности автомобиль – это не зонтик. Но несомненно и то, что его вполне можно использовать для защиты от дождя, хотя это будет очень ограниченное восприятие автомобиля. Точно так же, инкапсуляция – это не просто скрытие данных. Такое слишком узкое представление ограничивает возможности проектировщика.

Инкапсуляцию следует понимать как "любой вид скрытия". Другими словами, это механизм, способный скрывать данные. Но этот же механизм может использоваться для скрытия реализации, порожденных классов и многоного другого. Обратимся к диаграмме, представленной на рис. 8.1. Мы уже встречались с этой диаграммой в главе 7, *Шаблон Adapter*.

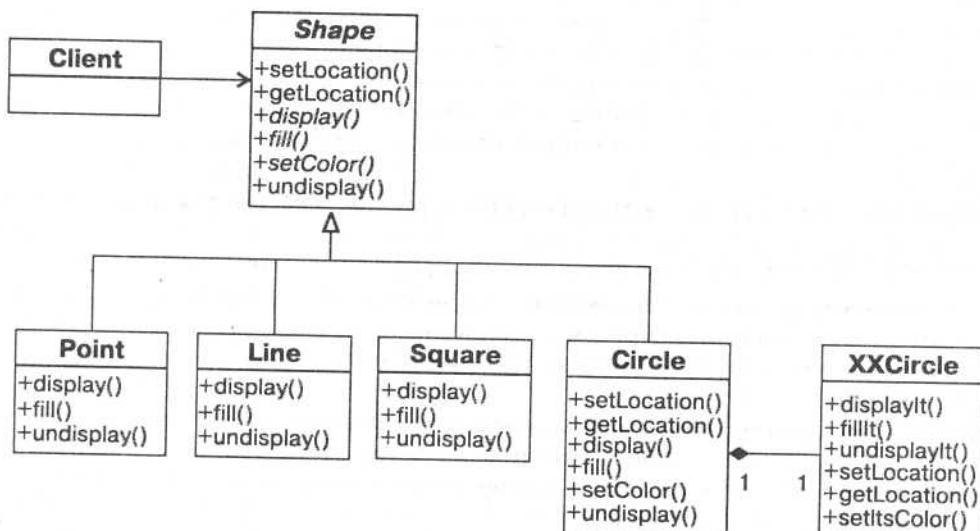


РИС. 8.1. Адаптация класса *XXCircle* с помощью класса *Circle*

На рис. 8.1 представлено несколько видов инкапсуляции.

- *Инкапсуляция данных.* Данные классов **Point**, **Line**, **Square** и **Circle** скрыты от всего остального мира.
- *Инкапсуляция методов.* Например, метод `setLocation()` класса **Circle** является скрытым.
- *Инкапсуляция подклассов.* Объектам-клиентам класса **Shape** классы **Point**, **Line**, **Square** или **Circle** не видны.
- *Инкапсуляция другого объекта.* Ничто в системе, кроме объектов класса **Circle**, не знает о существовании объектов класса **XXCircle**.

Следовательно, один тип инкапсуляция достигается созданием абстрактного класса, который демонстрирует полиморфное поведение, и клиент данного абстрактного класса не знает, с каким именно производным от него классом он имеет дело в действительности. Таким образом, адаптация интерфейса позволяет скрыть, что именно находится за адаптирующим объектом.

Преимущество подобного взгляда на инкапсуляцию состоит в том, что он упрощает разбиение (декомпозицию) программы. Инкапсулированные уровни в этом случае становятся интерфейсами, которые необходимо будет спроектировать. Инкапсуляция различных видов фигур в классе **Shape** позволяет добавлять в систему новые классы без модификации тех клиентских программ, которые их используют. Инкапсуляция объекта **XXCircle** в объекте **Circle** позволяет изменить его реализацию в будущем, если в этом возникнет потребность.

Когда объектно-ориентированная парадигма была впервые предложена, повторное использование классов понималось как одно из ее важнейших преимуществ. Обычно оно достигалось посредством разработки классов с последующим созданием на их основе новых, производных классов. Для процедуры создания этих подклассов, порожденных от других, базовых классов, использовался термин *специализация* (а для обратного перехода к базовому классу — *генерализация*).

Здесь я не собираюсь поднимать вопрос о точности этой терминологии, а просто рассматриваю тот подход, который в моем понимании представляет собой более мощный способ использования наследования. В приведенном выше примере можно было создать проект системы, основываясь на методе специализации класса **Shape** (с получением таких классов, как **Point**, **Line**, **Square** и **Circle**). Однако в этом случае мне, вероятно, не удалось бы спрятать указанные специализированные классы по отношению к методам использования класса **Shape**. Наоборот, скорее всего, я попытался бы воспользоваться преимуществами знания особенностей реализации каждого из производных классов.

Однако, если рассматривать класс **Shape** как обобщение для классов **Point**, **Line**, **Square** и **Circle**, то это упрощает восприятие их как единого понятия. В подобном случае разработчик, вероятнее всего, предпочтет разработать интерфейс и сделать класс **Shape** абстрактным. А это, в свою очередь, означает, что если потребуется ввести в систему поддержку новой фигуры, то необходимая модернизация существующего кода будет минимальна, поскольку никакой из объектов-клиентов не знает, с каким именно подтипом класса **Shape** он имеет дело.

Найдите то, что изменяется, и инкапсулируйте это

В своей книге¹ "банда четырех" предлагает следующее.

Выделите то, что будет изменяемым в вашем проекте. Этот подход противоположен методу, построенному на установлении причины, вынуждающей прибегнуть к переделке проекта. Вместо выявления тех причин, которые будут способны заставить нас внести изменения в проект, следует сосредоточиться на том, что мы хо-

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading MA: Addison-Wesley, 1995, с. 29.

тим уметь изменять, не переделывая проект. Идея состоит в том, что следует инкапсулировать те концепции, которые могут изменяться, — лейтмотив многих шаблонов проектирования.

Иными словами это можно выразить так: "Найдите изменяемое и инкапсулируйте это".

Подобное утверждение покажется странным, если понимать инкапсуляцию только как сокрытие данных. Но оно выглядит весьма разумным, если под инкапсуляцией понимать сокрытие конкретных классов с помощью абстрактных. Помещение в текст ссылок на абстрактный класс позволяет скрыть его возможные вариации.

В действительности, многие шаблоны проектирования используют инкапсуляцию для распределения объектов по различным уровням. Это позволяет разработчику вносить изменения на одном уровне, не оказывая влияния на другой. Этим достигается слабая связанность между объектами разных уровней.

Данный принцип очень важен для шаблона *Bridge* (мост), о котором речь пойдет в главе 9, *Шаблон Bridge*. Однако перед этим хотелось бы обсудить те предубеждения, которые присущи многим разработчикам.

Предположим, что мы работаем над проектом, заключающимся в моделировании различных характеристик животных. Требования к проекту состоят в следующем.

- Каждый тип животного имеет различное количество ног.
 - Представляющий животное объект должен быть способен запоминать и возвращать эту информацию.
- Каждый тип животного использует различный способ передвижения.
 - Представляющий животное объект должен быть способен возвращать сведения о том, сколько времени понадобится животному для перемещения из одного места в другое по земной поверхности заданного типа.

Типичный подход к представлению различного количества ног у животного состоит в том, что объект будет включать данное-член, содержащее соответствующее значение, а также два метода для записи и считывания этого значения. Однако для представления различного поведения обычно используется другой подход.

Предположим, что существует два различных метода передвижения животных: бег и полет. Поддержка этого требования потребует написания двух различных фрагментов программного кода — один для представления бега, а другой для представления полета. Вполне очевидно, что простой переменной в этом случае будет недостаточно. Требование реализации двух различных методов ставит нас перед необходимостью выбрать один из двух возможных подходов.

- Использование элемента данных, предназначенного для хранения сведений о типе передвижения животного, представленного объектом.
- Создание двух различных типов класса **Animal** (животное), производных от абстрактного класса **Animal**. Один из них будет предназначен для представления бегающих животных, а другой — летающих.

К сожалению, оба указанных подхода не лишены недостатков.

- Сильная связанность. Первый подход (использование флагка в совокупности с построенным на нем переключателем) ведет к сильной связанности, которая

отрицательно проявится, если флагку потребуется принимать какие-то дополнительные значения. В любом случае требуемый программный код будет довольно громоздким.

- **Илишняя детализация.** Второй подход требует жесткого присвоения подтипа класса **Animal** и не позволяет представлять в системе животных, способных и бегать, и летать одновременно.

Существует и третий вариант: пусть класс **Animal** включает объект, представляющий соответствующий способ передвижения (как показано на рис. 8.2).

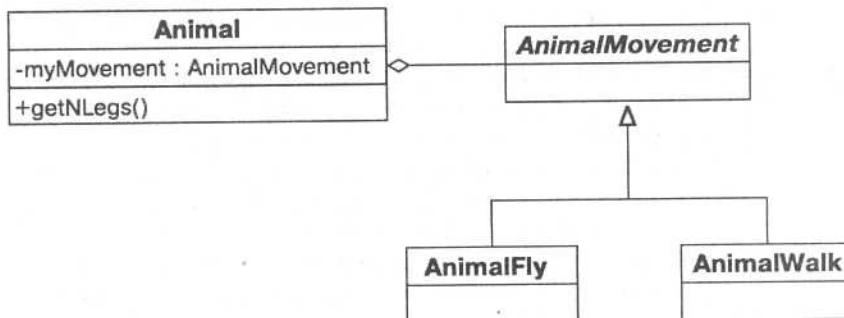


РИС. 8.2. Объект класса *Animal* содержит объект класса *AnimalMovement*

На первый взгляд такое решение кажется чрезмерно сложным. Однако в сущности оно совсем простое, поскольку предполагается лишь то, что класс **Animal** будет включать объект, представляющий способ передвижения животного. Подобный метод решения очень напоминает включение в класс данного-члена, содержащего сведения о количестве ног животного, – отличие лишь в том, что в этом случае сведения содержатся в данном-члене специфического объектного типа. Похоже, что на концептуальном уровне все это отличается сильнее, чем есть на самом деле, поскольку рис. 8.2 и 8.3 совершенно не похожи друг на друга.

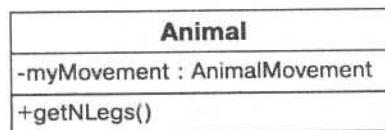


РИС. 8.3. Представление содержащего объекта как данного-члена

Многие разработчики полагают, что объект, содержащий другой объект, существенно отличается от объекта, содержащего только простые элементы данных. Однако данные-члены класса, которые обычно кажутся нам отличными от объектов (например, целые числа или числа двойной точности), на самом деле также являются объектами. В объектно-ориентированном программировании *все* является объектом, даже те встроенные типы данных, поведение которых является сугубо арифметическим.

Способы использования объектов для запоминания значений атрибутов и запоминания различных типов поведения фактически не отличаются друг от друга. Проще всего подтвердить это утверждение на конкретном примере. Предположим, необходимо создать систему управления сетью торговых точек. В этой системе должна обрабатываться приходная накладная. В накладной присутствует итоговая сумма. Для начала представим эту итоговую сумму как число с двойной точностью (тип `Double`). Однако если система предназначена для международной торговой сети, то очень скоро выяснится, что она обязательно должна обеспечивать конвертирование валют и другие подобные операции. Поэтому при расчетах потребуется использовать класс `Money` (деньги), включающий как значение суммы, так и описание типа валюты. Теперь для представления итоговой суммы в накладной необходимо будет использовать объект класса `Money`.

На первый взгляд использование класса `Money` в данном случае вызвано только необходимостью хранения дополнительных данных. Однако, когда потребуется конвертировать сумму в объекте класса `Money` из одного типа валюты в другой, именно данный объект `Money` должен будет выполнить такие преобразования, поскольку каждый объект должен нести ответственность сам за себя. Может показаться, что для выполнения преобразования достаточно просто добавить в класс еще одно данное-член, предназначенное для хранения коэффициента пересчета.

Однако в действительности все может оказаться сложнее. Например, от объекта может потребоваться выполнять конвертирование валюты по курсу прошлых периодов. В этом случае при расширении функциональных возможностей класса `Money` до класса `Currency` (валюта) мы по существу расширяем и функциональные возможности класса `SalesReceipt` (накладная), поскольку он включает в свой состав объекты класса `Money` (или класса `Currency`).

В последующих главах данная стратегия использования вложенных объектов для реализации требуемого поведения класса будет продемонстрирована при обсуждении нескольких новых шаблонов проектирования.

Общность и изменчивость в абстрактных классах

Обратимся к рис. 8.4, на котором отражены взаимосвязи между следующими концепциями.

- Анализ общности и анализ изменчивости.
- Концептуальный уровень, уровень спецификаций и уровень реализации.
- Абстрактный класс, его интерфейс и производные от него классы.

Как показано на рис. 8.4, анализ общности имеет отношение к концептуальному уровню проекта системы, тогда как анализ изменчивости относится к уровню его реализации, т.е. к процедурам конкретного воплощения системы.

Уровень спецификаций занимает промежуточное положение между двумя другими уровнями разработки системы. Поэтому он предусматривает выполнение обоих видов анализа. На уровне спецификаций определяется, как будет происходить взаимодействие с множеством объектов, которые концептуально схожи друг с другом, — т.е.

каждый из этих объектов представляет конкретный вариант некоторой общей концепции. На уровне реализации каждая такая общая концепция описывается абстрактным классом или интерфейсом.

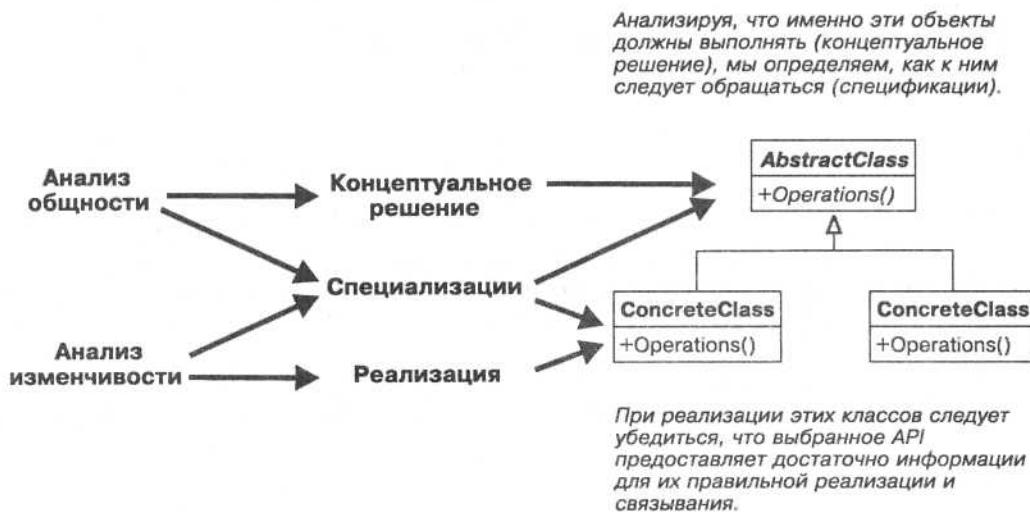


РИС. 8.4. Взаимосвязи между анализом общности/изменчивости, уровнями детализации и абстрактными классами

Новый взгляд на объектно-ориентированное проектирование кратко описывается в табл. 8.1.

Таблица 8.1. Новые концепции ООП

| Концепция ООП | Пояснение |
|---|--|
| Абстрактный класс → центральная объединяющая концепция | Абстрактный класс используется для представления базовой концепции, объединяющей все порожденные от него классы. Эта базовая концепция отражает некоторую общность, имеющую место в предметной области |
| Общность → какие абстрактные классы должны использоваться | Общности определяют абстрактные классы, которые должны существовать в системе |
| Вариации → классы, производные от абстрактного класса | Вариации, выявленные в пределах некоторой общности, представляются классами, производными от соответствующего абстрактного класса |
| Специализация → интерфейс для абстрактного класса | Интерфейс этих производных классов соответствует уровню спецификаций |

Подобный подход упрощает процесс проектирования классов и сводит его к процедуре из двух этапов, описанных в табл. 8.2.

Таблица 8.2. Проектирование классов

| Когда определяется... | Следует задаться вопросом... |
|-----------------------------------|---|
| Абстрактный класс (общность) | Какой интерфейс позволит обратиться ко всем обязательствам этого класса? |
| Порожденные классы (изменчивость) | Как выполнить данную конкретную реализацию (вариацию) исходя из имеющейся спецификации? |

Взаимосвязь между уровнем спецификаций и концептуальным уровнем можно описать так. *На уровне спецификаций определяется интерфейс, который необходим для реализации всех вариантов проявления (вариаций) некоторой концепции (общности), выявленной на концептуальном уровне.*

Взаимосвязь между уровнем спецификаций и уровнем реализации можно описать так. *Как, исходя из заданной спецификации, можно реализовать данное проявление (вариацию)?*

Резюме

Традиционное понимание концепций объекта, инкапсуляции и наследовании весьма ограниченно. Инкапсуляция представляет собой нечто большее, чем просто сокрытие данных. Расширение определения инкапсуляции до концепции сокрытия любых существующих категорий позволяет распределить объекты по разным уровням. Это, в свою очередь, позволяет вносить изменения на одном уровне, исключив нежелательное влияние этих действий на объекты другого уровня.

Наследование лучше использовать как метод определенной обработки различных конкретных классов, являющихся концептуально идентичными, а не как средство проведения специализации.

Концепция использования объектов для поддержки вариаций в поведении ничем не отличается от практики использования элементов данных для поддержки вариаций в значениях данных. Оба подхода предусматривают инкапсуляцию (т.е. расширение) как данных, так и поведения объектов.

Глава 9

Шаблон Bridge

Введение

Продолжим изучение шаблонов проектирования и рассмотрим шаблон Bridge (мост). Шаблон Bridge несколько сложнее тех шаблонов, о которых речь шла раньше, но и намного полезнее их.

В этой главе мы выполним следующее.

- Выведем концепцию шаблона Bridge, исходя из конкретного примера. Обсуждение будет проведено очень подробно, что должно помочь вам понять сущность этого шаблона.
- Рассмотрим ключевые особенности этого шаблона.
- Обсудим несколько поучительных примеров использования шаблона Bridge из моей личной практики.

Назначение шаблона проектирования Bridge

Согласно определению "банды четырех" "назначение шаблона Bridge состоит в отделении абстракции от реализации таким способом, который позволит им обеим изменяться независимо".¹

Я точно помню что впервые прочитав это определение, подумал:

Хм, ерунда какая-то!

А затем:

Я понимаю каждое слово в этом предложении, но не имею ни малейшего представления, что оно может означать.

Я знал следующее.

- *Отделение* означает обеспечение независимого поведения элементов, или, по крайней мере, явное указание на то, что между ними существуют некоторые отношения.
- *Абстракция* – это концептуальное представление о том, как различные элементы связаны друг с другом.

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 151.

Я также считал, что *реализация* – это конкретный способ представления абстракции, поэтому меня смущало предложение отделить абстракцию от конкретного способа ее реализации.

Как оказалось, мое замешательство было вызвано недопониманием того, что здесь в действительности представляет собой реализация. В данном случае под *реализацией* понимались те объекты, которые абстрактный и производные от него классы использовали для реализации своих функций (а не те классы, которые называются конкретными и являются производными от абстрактного класса). Но, честно говоря, даже если бы я все понимал правильно, не уверен, что это бы сильно мне помогло. Концепция, выраженная в приведенном выше определении, достаточно сложна, чтобы понять ее с первого раза.

Если вам сейчас тоже не до конца понятно, для чего предназначен шаблон Bridge, пусть это вас не беспокоит. Если же назначение этого шаблона вам вполне очевидно, то надо отдать должное вашей сообразительности.

Bridge – один из самых трудных для понимания шаблонов. До некоторой степени это вызвано тем, что он является очень мощным и часто применяется в самых различных ситуациях. Кроме того, он противоречит распространенной практике применения механизма наследования для реализации специальных случаев. Тем не менее, этот шаблон служит превосходным примером следования двум основным лозунгам технологии шаблонов проектирования: "Найди то, что изменяется, и инкапсулируй это" и "Компоновка объектов в структуру предпочтительней обращения к наследованию классов" (в дальнейшем мы убедимся в этом).

Описание шаблона проектирования Bridge на примере

Чтобы лучше понять идею построения шаблона Bridge и принципы его работы, рассмотрим конкретный пример поэтапно. Сначала обсудим установленные требования, а затем проанализируем вывод основной идеи шаблона и способы его применения.

Возможно, данный пример может показаться слишком простым. Однако присмотритесь к обсуждаемым в нем концепциям, а затем попытайтесь вспомнить аналогичные ситуации, с которыми нам приходилось сталкиваться ранее. Обратите особое внимание на следующее.

- Наличие вариаций в абстрактном представлении концепций.
- Наличие вариаций в том, как эти концепции реализуются.

Полагаю, у вас не появится сомнений, что приведенный ниже пример имеет много общего с обсуждавшейся выше задачей поддержки нескольких версий САПР. Однако мы не станем предварительно обсуждать все предъявляемые требования. Как правило, приступая к решению задачи, не удается сразу увидеть все существующие вариации.

Рекомендация. Формулируя требования к проекту, старайтесь как можно раньше и как можно чаще обдумывать, что в нем может изменяться в дальнейшем.

Предположим, что нам нужно написать программу, которая будет выводить изображения прямоугольников с помощью одной из двух имеющихся графических программ. Кроме того, допустим, что указания о выборе первой (Drawing Program 1 –

DP1) или второй (DP2) графической программы будут предоставляться непосредственно при инициализации прямоугольника.

Прямоугольники определяются двумя парами точек, как это показано на рис. 9.1. Различия между графическими программами описаны в табл. 9.1.

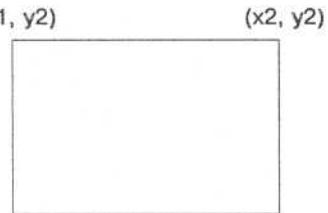


РИС. 9.1. Принцип описания прямоугольника

Таблица 9.1. Различия между двумя графическими программами

| Методы в DP1 | Методы в DP2 |
|--|---------------------------|
| Метод отображения линий draw_a_line(x1, y1, x2, y2) | drawline (x1, x2, y1, y2) |
| Метод отображения окружностей draw_a_circle(x, y, r) | drawcircle(x, y, r) |

Заказчик поставил условие, что объект коллекции (клиент, использующий программу отображения прямоугольников) не должен иметь никакого отношения к тому, какая именно графическая программа будет использоваться в каждом случае. Исходя из этого я пришел к заключению, что, поскольку при инициализации прямоугольника указывается, какая именно программа должна использоваться, можно создать два различных типа объекта прямоугольника. Один из них будет вызывать для отображения программу DP1, а другой — программу DP2. В каждом типе объекта будет присутствовать метод отображения прямоугольника, но реализованы они будут по-разному — как показано на рис. 9.2.

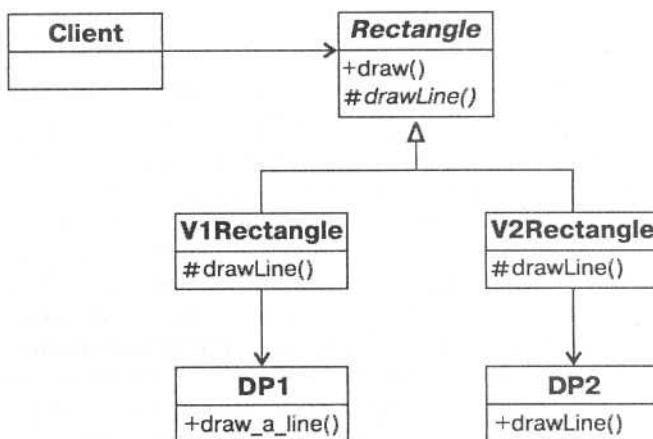


РИС. 9.2. Схема программы отображения прямоугольников с помощью программ DP1 и DP2

Создав абстрактный класс **Rectangle** (прямоугольник), можно воспользоваться тем преимуществом, что единственное несовпадение между различными типами его производных классов **Rectangle** состоит в способе реализации метода *drawLine()*. В классе **V1Rectangle** он реализован посредством ссылки на объект **DP1** и его метод *Draw_a_line()*, а в классе **V2Rectangle** – посредством ссылки на объект **DP2** и его метод *drawline()*. Таким образом, выбрав при инициализации требуемый тип объекта **Rectangle**, в дальнейшем можно полностью игнорировать эти различия. В листинге 9.1 приведен пример реализации этих объектов на языке Java.

Листинг 9.1. Пример реализации классов на языке Java. Версия 1

```
Class Rectangle{
    Public void draw () {
        DrawLine (_x1, _y1, _x2, _y1);
        DrawLine (_x2, _y1, _x2, _y2);
        DrawLine (_x2, _y2, _x1, _y2);
        DrawLine (_x1, _y2, _x1, _y1);
    }
    abstract protected void
        DrawLine (double x1, double y1,
                  double x2, double y2);
}

Class V1Rectangle extends Rectangle {
    DrawLine (double x1, double y1,
              double x2, double y2) {
        DP1.draw_a_line (x1, y1, x2, y2);
    }
}

Class V2Rectangle extends Rectangle {
    DrawLine (double x1, double y1,
              double x2, double y2) {
        // Аргументы в методе программы DP2 отличаются
        // и должны быть переупорядочены
        DP2.drawline (x1, x2, y1, y2);
    }
}
```

Предположим, что сразу после завершения и отладки кода, приведенного в листинге 9.1, на нас обрушивается одно из *трех неизбежных зол* (имеется в виду смерть, налоги и изменение требований). Необходимо включить в программу поддержку еще одного вида геометрических фигур – окружностей. Однако при этом уточняется, что объект коллекции (клиент) не должен ничего знать о различиях, существующих между объектами, представляющими прямоугольники (**Rectangle**) и окружности (**Circle**).

Логично будет сделать вывод, что можно применить выбранный ранее подход и просто добавить еще один уровень в иерархию классов программы. Потребуется только добавить в проект новый абстрактный класс (назовем его **Shape** (фигура)), а классы **Rectangle** и **Circle** будут производными от него. В этом случае объект **Client** сможет просто обращаться к объекту класса **Shape**, совершенно не заботясь о том, какая именно геометрическая фигура им представлена.

Для аналитика, только начинающего работать в области объектно-ориентированной разработки, такое решение может показаться вполне естественным – реализовать изменение в требованиях только с помощью механизма наследования. Начав

со схемы, представленной на рис. 9.2, добавим к ней новый уровень с классом *Shape*. Затем для всех видов фигур организуем их реализацию с помощью каждой из графических программ, создав по отдельному производному классу для вызова объектов *DP1* и *DP2*, как в классе *Rectangle*, так и в классе *Circle*. В итоге будет получена схема, подобная представленной на рис. 9.3.

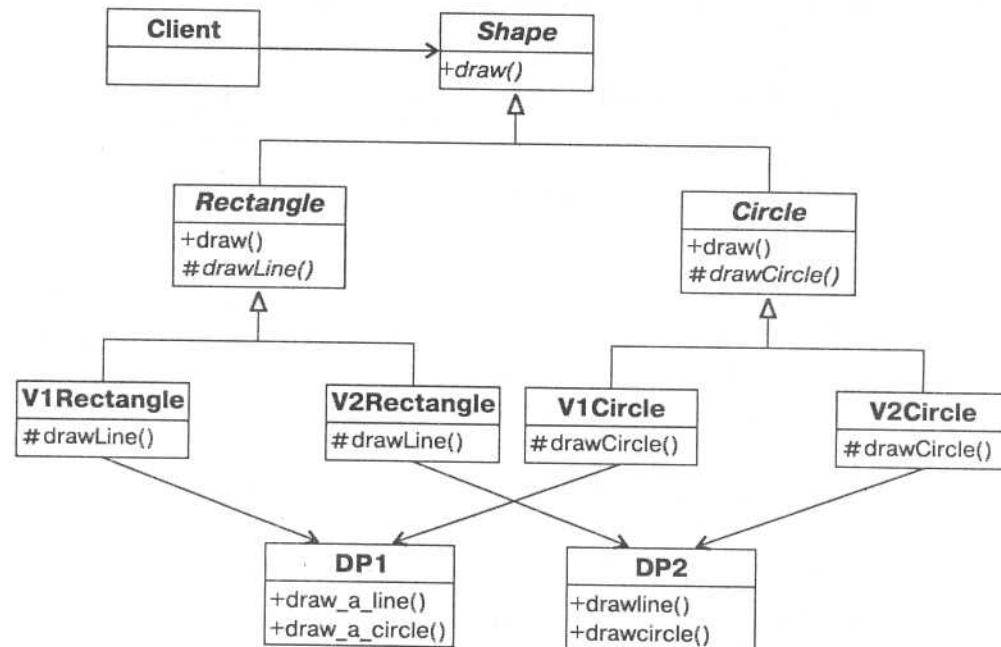


РИС. 9.3. Простейший прямолинейный подход к реализации двух фигур с помощью двух графических программ

Класс *Circle* можно реализовать тем же способом, что и класс *Rectangle* — как показано в листинге 9.2. Единственное отличие состоит в том, что вместо метода *drawLine()* используется метод *drawCircle()*.

Листинг 9.2. Пример реализации классов на языке Java. Версия 2

```

abstract class Shape {
    abstract public void draw ();
}

abstract class Rectangle extends Shape {
    public void draw () {
        drawLine(_x1, _y1, _x2, _y1);
        drawLine(_x2, _y1, _x2, _y2);
        drawLine(_x2, _y2, _x1, _y2);
        drawLine(_x1, _y2, _x1, _y1);
    }
}

abstract protected void
drawLine()
  
```

```

        double x1, double y1,
        double x2, double y2);
    }

    class V1Rectangle extends Rectangle {
        protected void drawLine (
            double x1, double y1,
            double x2, double y2) {
            DP1.draw_a_line(x1, y1, x2, y2);
        }
    }

    class V2Rectangle extends Rectangle {
        protected void drawLine (
            double x1, double x2,
            double y1, double y2) {
            DP2.drawline(x1, x2, y1, y2);
        }
    }

    abstract class Circle {
        public void draw () {
            drawCircle(x, y, r);
        }
        abstract protected void
        drawCircle (
            double x, double y, double r);
    }

    class V1Circle extends Circle {
        protected void drawCircle() {
            DP1.draw_a_circle(x, y, r);
        }
    }

    class V2Circle extends Circle {
        protected void drawCircle() {
            DP2.drawcircle(x, y, r);
        }
    }
}

```

Для того чтобы лучше понять особенности этого проекта, рассмотрим конкретный пример. Например, проанализируем, как работает метод `draw()` класса `V1Rectangle`.

- Метод `draw()` класса `Rectangle` не изменился (в нем четыре раза подряд вызывается метод `drawLine()`).
- Метод `drawLine()` реализуется посредством обращения к методу `draw_a_line()` объекта `DP1`.

Схематично все это представлено на рис. 9.4.

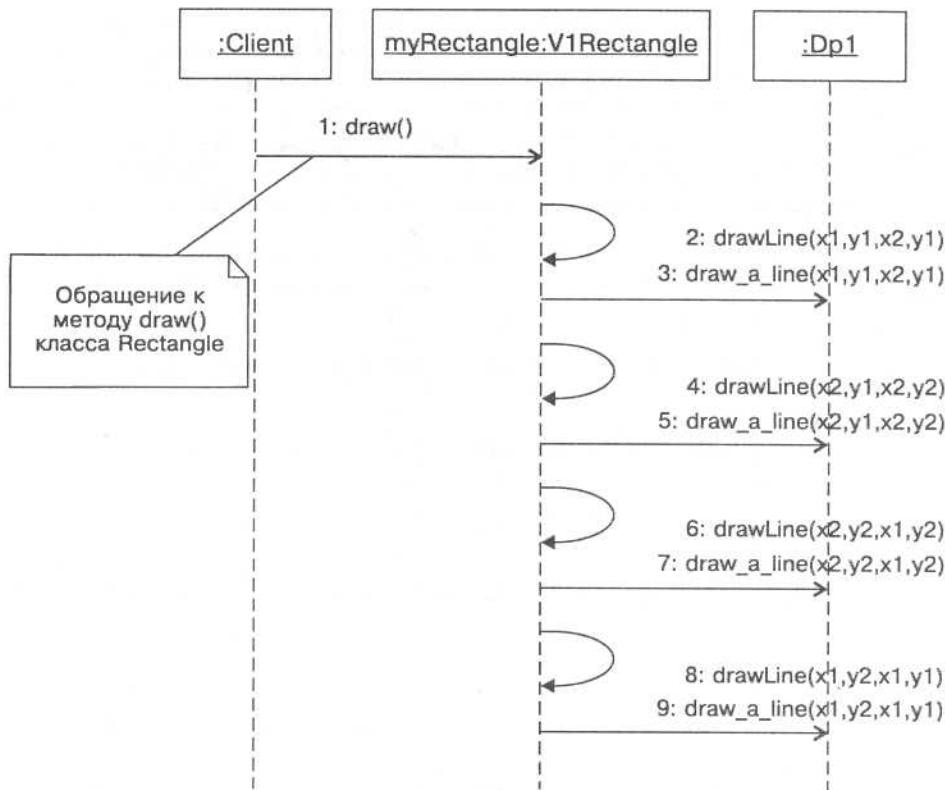


РИС. 9.4. Диаграмма последовательностей обращения к объекту класса V1Rectangle

Чтение диаграмм последовательностей

Выше, в главе 2, мы уже обсуждали язык UML — унифицированный язык моделирования. Диаграмма на рис. 9.4 представляет собой один из видов диаграмм взаимодействия, называемый *диаграммой последовательностей*. Это типичная диаграмма языка UML. Ее назначение состоит в том, чтобы продемонстрировать взаимодействие объектов в системе.

- Каждый прямоугольник вверху диаграммы обозначает объект. Он может быть поименован или нет.
- Если у объекта есть имя, оно указывается слева от двоеточия.
- Класс, которому принадлежит объект, указывается справа от двоеточия. Например, на рис. 9.4 имя среднего объекта myRectangle, и этот объект представляет собой экземпляр класса V1Rectangle.

Диаграмму последовательностей следует читать сверху вниз. На диаграмме каждое пронумерованное предложение представляет собой сообщение, отправленное объектом самому себе или другому объекту.

- Последовательность начинается с непоименованного объекта класса Client, вызывающего метод draw() объекта myRectangle класса V1Rectangle.

- Этот метод четыре раза вызывает метод `drawLine()` собственного объекта (этапы 2, 4, 6 и 8 на схеме). Обратите внимание на изогнутую стрелку, указывающую на оси времени на сам объект `myRectangle`.
- В свою очередь, метод `drawLine()` каждый раз вызывает метод `draw_a_line()` непоименованного объекта класса `DP1` (этапы 3, 5, 7 и 9 на диаграмме).

Несмотря на то что при взгляде на диаграмму классов складывается впечатление о наличии на ней множества объектов, в действительности, мы имеем дело только с тремя объектами, как показано на рис. 9.5.

- Объект класса `Client`, потребовавший отобразить прямоугольник.
- Объект класса `V1Rectangle`.
- Объект класса `DP1`, представляющий графическую программу.

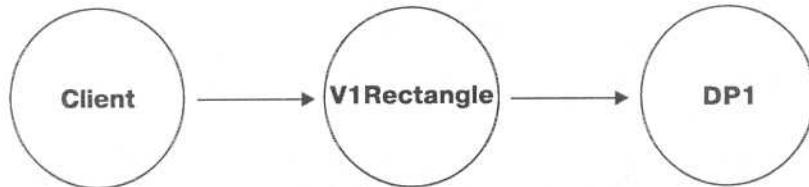


РИС. 9.5. Объекты, представленные на диаграмме последовательностей

Когда объект-клиент посыпает сообщение объекту класса `V1Rectangle` (с именем `myRectangle`) с требованием выполнить метод `draw()`, последний реализует метод `draw()` класса `Rectangle`, выполняя этапы со 2 по 9, показанные на диаграмме.

К сожалению, этот подход создает новые проблемы. Взгляните еще раз на рис. 9.3 и обратите внимание на третий ряд классов. Глядя на них, можно сделать следующие выводы.

- Классы в этом ряду представляют те четыре конкретных подтипа класса `Shape`, с которыми мы имеем дело.
- Что произойдет, если появится третий тип графической программы, т.е. еще один возможный вариант реализации графических функций? В этом случае потребуется создать шесть различных подтипов класса `Shape` (для представления двух видов фигур и трех графических программ).
- Теперь предположим, что требуется реализовать поддержку нового, третьего, типа фигур. В этом случае нам понадобятся уже девять различных подтипов класса `Shape` (для представления трех видов фигур и трех графических программ).

Быстрый рост количества требуемых производных классов вызван тем, что в данном решении абстракция (виды фигур) и реализация (графические программы) жестко связаны между собой. Каждый класс, представляющий конкретный тип фигуры, должен точно знать, какую из существующих графических программ он использует. Для решения данной проблемы необходимо отделить изменения в абстракции от из-

менений в реализации таким образом, чтобы количество требуемых классов возрастало линейно – как показано на рис. 9.6.

Вот мы и пришли к приведенному в начале главы определению назначения шаблона Bridge – отделение абстракции от реализации, выполненное таким способом, который позволит им обеим изменяться независимо².



РИС. 9.6. Шаблон Bridge отделяет изменения в абстракции от изменений в реализации

Перед тем как рассмотреть возможное решение проблемы и вывести из него шаблон Bridge, остановимся на нескольких других проблемах (помимо комбинаторного взрыва количества классов).

Еще раз посмотрим на рис. 9.3 и зададимся вопросом: какие еще недостатки имеет данный проект?

- Существует ли в нем избыточность?
- Что характерно для данного проекта – сильная или слабая связность?
- Сильно или слабо связаны между собой отдельные элементы проекта?
- Согласились бы вы выполнять сопровождение программ этого проекта в будущем?

Злоупотребление наследованием

Будучи начинающим разработчиком объектно-ориентированных проектов, я обычно решал проблемы обсуждаемого типа, выделяя их как специальные случаи с последующим применением механизма наследования. Мне нравилась сама идея наследования, поскольку это представлялось мне новым и мощным подходом. Я использовал наследование всегда, когда это было возможным. Такой подход выглядит вполне естественным для новичка, но, в сущности, он весьма наивен.

К сожалению, многие из существующих методик обучения объектно-ориентированному проектированию фокусируют главное внимание на абстракции данных, что делает проектирование чрезмерно зависимым от внутренней организации объектов. И даже когда я стал опытным разработчиком, то еще долго сохранял верность парадигме проектирования, основанной на широком использовании механизма наследования. Иными словами, я определял характеристики создаваемых классов исходя из особенностей их реализации. Однако характеристики объектов

² Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 151.

должны определяться их обязательствами, а не их содержанием или внутренней структурой. Скажем, объекты могут отвечать за предоставление информации о самих себе. Например, объекту, представляющему в системе покупателя, может потребоваться умение сообщать его имя. Следует стремиться думать об объектах прежде всего в терминах их обязательств, а не внутренней структуры.

Опытные разработчики объектно-ориентированных проектов пришли к выводу, что для полной реализации всей мощи механизма наследования следует применять его выборочно. Использование шаблонов проектирования помогает быстрее это понять. Оно позволяет перейти от выделения отдельной специализации для каждого изменения (методом наследования) к перемещению этих изменений в используемые объекты сторонней или собственной разработки.

Поначалу, столкнувшись с указанными выше проблемами, я решил, что причина их появления кроется в неправильном построении иерархии наследования. Поэтому я попробовал применить альтернативный вариант иерархии классов, представленный на рис. 9.7.

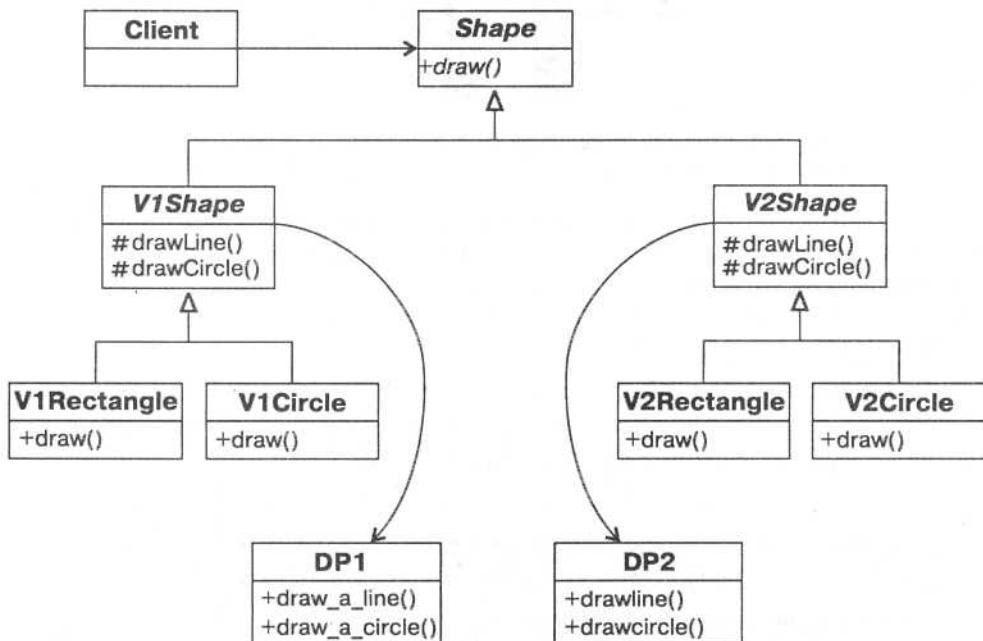


РИС. 9.7. Альтернативный вариант иерархии наследования классов

Здесь по-прежнему используются те же четыре класса, представляющие все возможные комбинации. Однако в этой версии устранена избыточность пакетов классов для графических программ **DP1** и **DP2**.

К сожалению, в этом варианте мне так и не удалось устраниТЬ избыточность, связанную с наличием двух типов классов **Rectangle** и двух типов классов **Circle**, причем в каждой паре родственных классов используется один и тот же метод **draw()**.

Как бы там ни было, исключить в новой схеме упомянутое выше комбинаторное увеличение количества классов так и не удалось.

Диаграмма последовательностей для нового решения представлена на рис. 9.8.

Хотя по сравнению с первоначальным решением новый вариант иерархии классов действительно обладает определенными преимуществами, он все же имеет недостатки. Здесь по-прежнему сохраняются проблемы слабой связности классов и их избыточной связанности между собой.

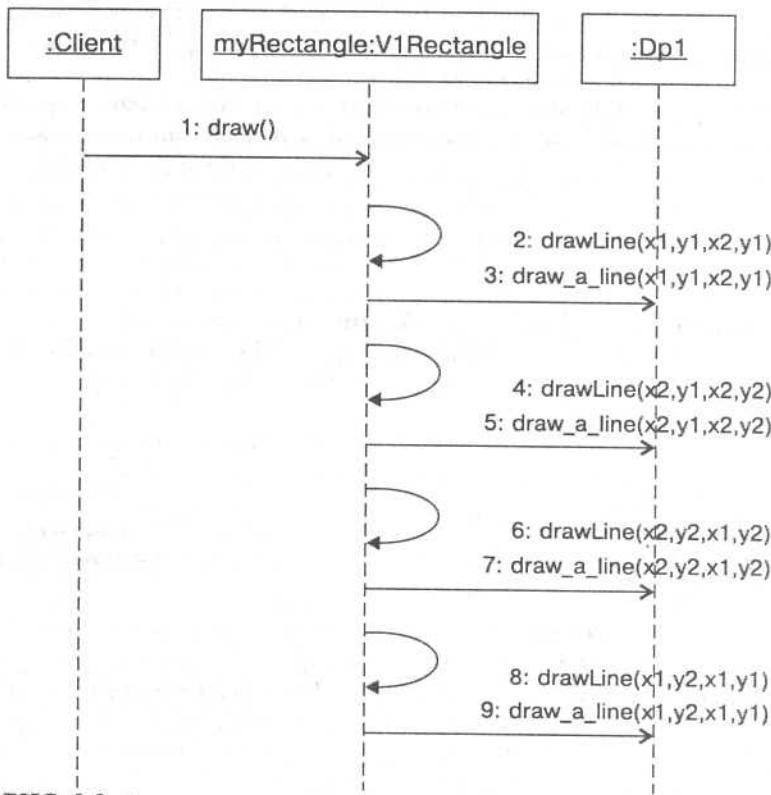


РИС. 9.8. Диаграмма последовательностей для нового варианта иерархии классов

Замечание. Эта версия иерархии классов по-прежнему не вызывает желания сопровождать программный код системы! Должно существовать лучшее решение.

Ищите альтернативы на начальной стадии проектирования

Хотя предложенный здесь альтернативный вариант иерархии наследования классов немного лучше первоначального, следует отметить, что сама по себе практика поиска альтернативы первоначальному варианту проекта — это хороший стиль разработки. Слишком часто проектировщики останавливаются на том варианте, который первый пришел им в голову, и в дальнейшем не предпринимают каких-либо попыток его улучшить. Я не призываю вас к чрезмерно глубокому и исчерпывающему изучению всех возможных альтернатив (это еще один способ ввернуть проект в состояние "паралича за счет анализа"). Однако сделать шаг назад и подумать, как можно было бы преодолеть затруднения, свойственные исходному варианту проекта, — это верный подход. Фактически, именно подобный шаг назад и отказ от дальнейшей работы над заведомо слабым проектным решением привели меня к пониманию всей моей подхода с использованием шаблонов проектирования, описанию которого, собственно, и посвящена эта книга.

В архитектурном смысле анализ общности дает архитектуре ее долговечность, а анализ изменчивости способствует достижению удобства в использовании.

Другими словами, если изменчивость — это особые случаи в рамках заданной предметной области, то общность устанавливает в ней концепции, объединяющие эти особые случаи между собой. Общие концепции будут представлены в системе абстрактными классами. Вариации, обнаруженные при анализе изменчивости, реализуются посредством создания конкретных классов, производных от этих абстрактных классов.

В объектно-ориентированном проектировании стала уже почти аксиомой практика, когда разработчик, анализируя описание проблемной области, выделяет в нем существительные и создает объекты, представляющие их. Затем он отыскивает глаголы, связанные с этими существительным (т.е. их действия), и реализует их, добавляя к объектам необходимые методы. Подобный процесс проявления повышенного внимания к существительным и глаголам в большинстве случаев приводит к созданию слишком громоздкой иерархии классов. Я считаю, что анализ общности и изменчивости как первичный инструмент выделения объектов, в действительности, предпочтительнее поиска существительных и глаголов в описании предметной области. (Полагаю, что книга Джима Коплина убедительно подтверждает эту точку зрения.)

В практике проектирования для работы с изменяющимися элементами применяются две основные стратегии.

- Найти то, что изменяется, и инкапсулировать это.
- Преимущественно использовать композицию вместо наследования.

Ранее для координации изменяющихся элементов разработчики часто создавали обширные схемы наследования классов. Однако вторая из приведенных выше стратегий рекомендует везде, где только возможно, заменять наследование композицией. Идея состоит в том, чтобы инкапсулировать изменения в независимых классах, что позволит при обработке будущих изменений обойтись без модификации программного кода. Одним из способов достижения подобной цели является помещение каждого подверженного изменениям элемента в собственный абстрактный класс с последующим анализом, как эти абстрактные классы соотносятся друг с другом.

Внимательный взгляд на инкапсуляцию

Чаще всего начинающих разработчиков объектно-ориентированных систем учат, что инкапсуляция заключается в скрытии данных. К сожалению, это очень ограниченное определение. Несомненно, что инкапсуляция действительно позволяет скрывать данные, но она может использоваться и для многих других целей. Еще раз обратимся к рис. 7.2, на котором показано, как инкапсуляция применяется на нескольких уровнях. Безусловно, с ее помощью скрываются данные для каждой конкретной фигуры. Однако обратите внимание на то, что объект *Client* также ничего не знает о конкретных типах фигур. Следовательно, объект *Client* не имеет сведений о том, что объекты класса *Shape*, с которыми он взаимодействует, в действительности являются объектами разных конкретных классов — *Rectangle* и *Circle*. Таким образом, тип конкретного класса, с которым объект *Client* взаимодействует, скрыт от него (инкапсулирован). Это тот вид инкапсуляции, который подразумевается во фразе "найдите то, что изменяется, и инкапсулируйте это". Здесь, как раз было обнаружено то, что изменяется, и оно было скрыто за "стеной" абстрактного класса (см. главу 8, *Расширение горизонтов*).

Рассмотрим данный процесс на примере задачи с вычерчиванием прямоугольника. Сначала идентифицируем то, что изменяется. В нашем случае это различные типы фигур (представленных абстрактным классом **Shape**) и различные версии графических программ. Следовательно, общими концепциями являются понятия типа фигуры и графической программы, как показано на рис. 9.9. (Обратите внимание на то, что имена классов выделены курсивом, поскольку это абстрактные классы.)

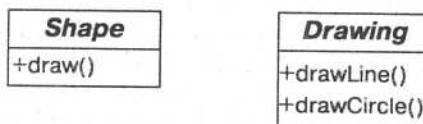


РИС. 9.9. Сущности, которые изменяются в нашем примере

В данном случае предполагается, что абстрактный класс **Shape** инкапсулирует концепцию типов фигур, с которыми необходимо работать. Каждый тип фигуры должен знать, как себя нарисовать. В свою очередь, абстрактный класс **Drawing** (рисование) отвечает за вычерчивание линий и окружностей. На рисунке указанные выше обязательства представлены посредством определения соответствующих методов в каждом из классов.

Теперь необходимо представить на схеме те конкретные вариации, с которыми мы будем иметь дело. Для класса **Shape** это прямоугольники (класс **Rectangle**) и окружности (класс **Circle**). Для класса **Drawing** это графические программы DP1 (класс **V1Drawing**) и DP2 (класс **V2Drawing**). Все это схематически показано на рис. 9.10.

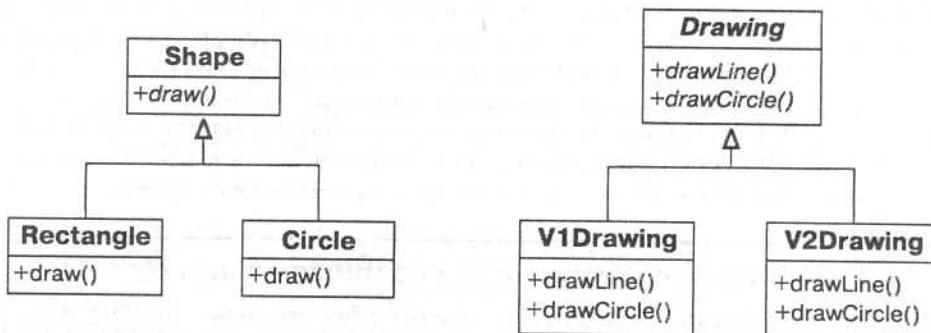


РИС. 9.10. Представление конкретных вариаций

Сейчас наша диаграмма имеет очень упрощенный вид. Определенно известно, что класс **V1Drawing** будет использовать программу DP1, а класс **V2Drawing** – программу DP2, но на схеме еще не указано, как это будет сделано. Пока мы просто описали концепции, присутствующие в проблемной области (фигуры и графические программы), и указали их возможные вариации.

Имея перед собой два набора классов, очевидно, следует задаться вопросом, как они будут взаимодействовать друг с другом. На этот раз попытаемся обойтись без того, чтобы добавлять в систему еще один новый набор классов, построенный на углублении иерархии наследования, поскольку последствия этого нам уже известны.

(см. рис. 9.3 и 9.7). На этот раз мы подойдем с другой стороны и попробуем определить, как эти классы могут использовать друг друга (в полном соответствии с приведенным выше утверждением о предпочтительности композиции над наследованием). Главный вопрос здесь состоит в том, какой же из абстрактных классов будет использовать другой?

Рассмотрим две возможности: либо класс **Shape** использует классы графических программ, либо класс **Drawing** использует классы фигур.

Начнем со второго варианта. Чтобы графические программы могли рисовать различные фигуры непосредственно, они должны знать некоторую общую информацию о фигурах: что они собой представляют и как выглядят. Однако это требование нарушает фундаментальный принцип объектной технологии: каждый объект должен нести ответственность только за себя.

Это требование также нарушает инкапсуляцию. Объекты класса **Drawing** должны были бы знать определенную информацию об объектах класса **Shape**, чтобы иметь возможность отобразить их (а именно – тип конкретной фигуры). В результате объекты класса **Drawing** фактически оказываются ответственными не только за свое собственное поведение.

Вернемся к первому варианту. Что если объекты класса **Shape** для отображения себя будут использовать объекты класса **Drawing**? Объектам класса **Shape** не нужно знать, какой именно тип объекта класса **Drawing** будет использоваться, поэтому классу **Shape** можно разрешить ссылаться на класс **Drawing**. Дополнительно класс **Shape** в этом случае можно сделать ответственным за управление рисованием.

Последний вариант выглядит предпочтительнее. Графически это решение представлено на рис. 9.11.

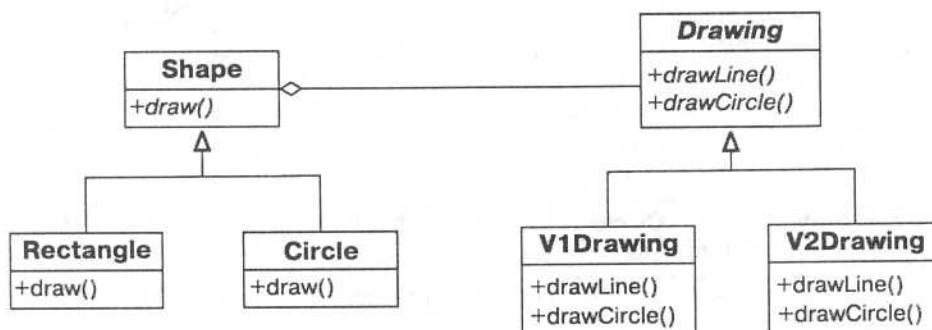


РИС. 9.11. Связывание абстрактных классов между собой

В данном случае класс **Shape** использует класс **Drawing** для проявления собственного поведения. Мы оставляем без внимания особенности реализации классов **V1Drawing**, использующего программу DP1, и **V2Drawing**, использующего программу DP2. На рис. 9.12 эта задача решена посредством добавления в класс **Shape** защищенных методов `drawLine()` и `drawCircle()`, которые вызывают методы `drawLine()` и `drawCircle()` объекта **Drawing**, соответственно.

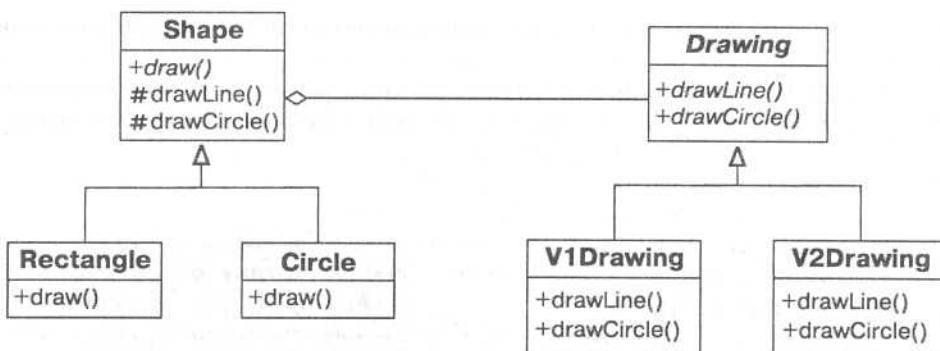


РИС. 9.12. Расширение проекта

Одно правило, одно место

При проектировании реализации очень важно следовать стратегии, согласно которой каждое правило реализуется только в одном месте. Другими словами, если имеется некоторое правило, определяющее способ выполнения каких-либо действий, оно должно быть реализовано только в одном месте. Такой подход обычно приводит к получению программного кода с большим количеством коротких методов. При этом за счет минимальных дополнительных усилий устраняется дублирование и удается избежать множества потенциальных проблем. Дублирование вредно не только из-за выполнения дополнительной работы по многократному вводу одинаковых фрагментов кода, но и в большей степени из-за вероятности того, что при каких-либо изменениях в будущем модификация будет произведена не везде.

В методе `draw()` класса `Rectangle` можно было бы непосредственно вызвать метод `DrawLine()` того объекта класса `Drawing`, к которому обращается объект класса `Shape`. Однако, следуя указанной выше стратегии, можно улучшить программный код, создав в классе `Shape` метод `drawLine()`, который и будет вызывать метод `DrawLine()` класса `Drawing`.

Я не считаю себя консерватором (по крайней мере, в большинстве случаев), но в этом вопросе я полагаю совершенно необходимым всегда соблюдать установленные правила. В примере, который будет обсуждаться ниже, класс `Shape` содержит метод `drawLine()`, описывающий правило вычерчивания линии объектом `Drawing`. Аналогичный метод `drawCircle()` предназначен для отображения окружностей. Следуя рекомендуйм здесь стратегии, мы готовим фундамент для появления других производных классов фигур, при вычерчивании которых могут потребоваться линии и окружности.

Где впервые была предложена стратегия реализации каждого правила в одном месте? Хотя многие упоминают о ней в своих публикациях, она утвердилась в фольклоре разработчиков объектно-ориентированных проектов уже давно и всегда представлялась как оптимальная практика проектирования. Совсем недавно Кент Бекк (Kent Beck) назвал эту стратегию правилом "однажды и только однажды"⁶.

Он определяет это правило так.

- Система (и код, и тесты) должна иметь доступ ко всему, к чему, по вашему мнению, она должна иметь доступ.
- Система не должна содержать никакого дублирующегося кода.

Эти две составляющие вместе и представляют собой правило "однажды и только однажды".

⁶ Beck K. Extreme Programming Explained: Embrace Change, Reading, MA: Addison Wesley, 2000, с. 108, 109.

На рис. 9.13 показано, как абстракция **Shape** может быть отделена от реализации **Drawing**.

С точки зрения методов, новый вариант системы весьма похож на реализацию, построенную на наследовании (см. рис. 9.3). Главное отличие состоит в том, что здесь методы расположены в различных объектах.

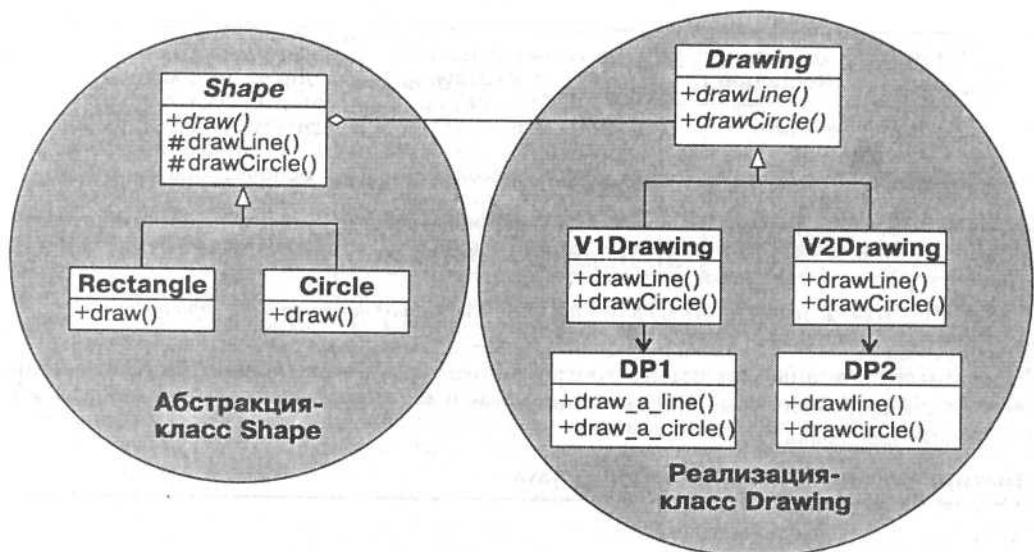


РИС. 9.13. Диаграмма классов, иллюстрирующая отделение абстракции от реализации

Как уже было сказано в начале этой главы, мое замешательство при первом знакомстве с определением шаблона Bridge было вызвано неправильным пониманием термина "реализация". Понапалу я полагал, что этот термин относится к тому, как конкретная абстракция реализуется в программном коде.

Шаблон Bridge позволяет взглянуть на реализацию как на нечто, находящееся вне наших объектов, нечто *используемое* этими объектами. В результате мы получаем намного большую свободу за счет сокрытия вариации в реализации от вызывающей части программы. Разрабатывая объекты по этому принципу, я также обнаружил возможность размещения вариаций различного типа в раздельных иерархиях классов. Иерархия, изображенная на рис. 9.13 слева, включает вариации в абстракциях. Иерархия справа включает вариации в том, как будут реализованы эти абстракции. Такой подход отвечает новой парадигме создания объектов (использование анализа общности и изменчивости), упомянутой выше.

Очень легко визуализировать сказанное, если вспомнить, что в любой момент времени в системе существует только три взаимодействующих объекта – несмотря на то, что в ней реализовано около десятка различных классов (рис. 9.14).



В действительности это объект **Rectangle** или **Circle**, но объект **Client** не может установить, какой именно, поскольку для него они выглядят совершенно одинаково

В действительности это объект **V1Drawing** или **V2Drawing**, но объект **Shape** не может установить, какой именно, поскольку для него они выглядят совершенно одинаково

Это должен быть объект того класса, который соответствует данному конкретному случаю, и использующий его объект **Drawing** должен знать, какой это объект

РИС. 9.14. В любой момент времени в программе существует только три объекта

Достаточно полный фрагмент программного кода для нашего примера представлен в листинге 9.3 для языка Java и в листингах 9.4–9.6, помещенных в приложение к этой главе, для языка C++.

Листинг 9.3. Фрагмент кода на языке Java

```

class Client {
    public static void main
        (String argv[]) {
            Shape r1, r2;
            Drawing dp;

            dp = new V1Drawing();
            r1 = new Rectangle(dp, 1, 1, 2, 2);

            dp = new V2Drawing();
            r2 = new Circle(dp, 2, 2, 3);

            r1.draw();
            r2.draw();
        }
}

abstract class Shape {
    abstract public draw() ;
    private Drawing _dp;

    Shape (Drawing dp) {
        _dp= dp;
    }

    public void drawLine (
        double x1, double y1,
        double x2, double y2) {
        _dp.drawLine(x1, y1, x2, y2);
    }

    public void drawCircle (
    
```

```
    double x, double y, double r) {
        _dp.drawCircle(x, y, r);
    }
}

abstract class Drawing {
    abstract public void drawLine (
        double x1, double y1,
        double x2, double y2);
    abstract public void drawCircle (
        double x, double y, double r);
}

class V1Drawing extends Drawing {
    public void drawLine (
        double x1, double y1,
        double x2, double y2) {
        DP1.draw_a_line(x1, y1, x2, y2);
    }
    public void drawCircle (
        double x, double y, double r) {
        DP1.draw_a_circle(x, y, r);
    }
}
class V2Drawing extends Drawing {
    public void drawLine (
        double x1, double y1,
        double x2, double y2) {
        // В программе DP2 порядок аргументов отличается
        // и они должны быть перестроены
        DP2.drawline(x1, x2, y1, y2);
    }
    public void drawCircle (
        double x, double y, double r) {
        DP2.drawcircle(x, y, r);
    }
}

class Rectangle extends Shape {
    public Rectangle (
        Drawing dp,
        double x1, double y1,
        double x2, double y2) {
        super(dp);
        _x1 = x1; _x2 = x2;
        _y1 = y1; _y2 = y2;
    }

    public void draw () {
        drawLine(_x1, _y1, _x2, _y1);
        drawLine(_x2, _y1, _x2, _y2);
        drawLine(_x2, _y2, _x1, _y2);
        drawLine(_x1, _y2, _x1, _y1);
    }
}

class Circle extends Shape {
    public Circle (
        Drawing dp,
        double x, double y, double r) {
        super(dp);
        _x = x; _y = y; _r = r ;
    }
}
```

```

public void draw () {
    drawCircle(_x, _y, _r);
}
}

// Ниже приведена упрощенная реализация
// для классов DP1 and DP2

class DP1 {
    static public void draw_a_line (
        double x1, double y1,
        double x2, double y2) {
        // Реализация
    }
    static public void draw_a_circle(
        double x, double y, double r) {
        // Реализация
    }
}

class DP2 {
    static public void drawline (
        double x1, double x2,
        double y1, double y2) {
        // Реализация
    }
    static public void drawcircle (
        double x, double y, double r) {
        // Реализация
    }
}

```

Особенности использования шаблона Bridge

Теперь, когда мы проанализировали, как шаблон Bridge работает, стоит посмотреть на него с более концептуальной точки зрения. Как показано на рис. 9.13, шаблон включает две части – абстрактную (со своими производными классами) и реализаций. При проектировании с использованием шаблона Bridge полезно всегда помнить об этих двух частях. Интерфейс в части реализации следует разрабатывать с учетом особенностей различных производных классов того абстрактного класса, который этот интерфейс будет поддерживать. Обратите внимание на то, что проектировщик не обязательно должен помещать в интерфейс реализации всех возможных производных классов абстрактного класса (это еще одна возможная причина возникновения "паралича от анализа"). Следует принимать во внимание только те производные классы, которые действительно необходимо поддерживать. Не раз и не два авторы этой книги получали подтверждение, что даже небольшое усилие по увеличению гибкости в этой части проекта существенно его улучшает.

Замечание. В языке C++ реализация шаблона Bridge должна осуществляться только с помощью абстрактного класса, определяющего открытый интерфейс. В языке Java могут использоваться как абстрактный класс, так и интерфейс. Выбор зависит от того, дает ли преимущество разделение в реализации общих черт абстрактных классов. (Подробности – в книге "Peter Coad, Java Design", описание которой можно найти в главе 22.)

Дополнительные замечания о шаблоне Bridge

Обратите внимание на то, что решение, представленное на рис. 9.12 и 9.13, объединяет шаблоны Adapter и Bridge. Это сделано из-за того, что графические программы, которые необходимо использовать, были заранее жестко заданы. В этих программах уже есть интерфейсы, с которыми мы вынуждены работать. Поэтому, чтобы можно было работать с ними одинаково, необходимо воспользоваться шаблоном Adapter.

Несмотря на то что на практике шаблон Adapter зачастую оказывается включенным в шаблон Bridge, он, тем не менее, вовсе не является частью шаблона Bridge.

Основные характеристики шаблона Bridge

| | |
|----------------|--|
| Назначение | Отделение набора объектов реализаций от набора объектов, использующих их |
| Задача | Классы, производные от абстрактного класса, должны использовать множество классов реализации этой абстракции, не вызывая при этом лавинообразного увеличения количества классов в системе |
| Способ решения | Определение интерфейса для всех используемых версий реализации и использование его во всех классах, порожденных от абстрактного |
| Участники | Класс <i>Abstraction</i> определяет интерфейс для объектов, представляющих сторону реализации. Класс <i>Implementor</i> определяет интерфейс для конкретных классов реализации. Классы, производные от класса <i>Abstraction</i> , используют классы, производные от класса <i>Implementor</i> , не интересуясь тем, с каким именно классом <i>ConcreteImplementorX</i> они имеют дело |
| Следствия | Отделение классов реализаций от объектов, которые их используют, упрощает расширение системы. Объекты-клиенты ничего не знают об особенностях конкретных реализаций |
| Реализация | <ul style="list-style-type: none"> Инкапсуляция вариантов реализации в абстрактном классе. Включение методов работы с ним в базовый абстрактный класс, представляющий абстракцию, подлежащую реализации. <p><i>Замечание.</i> В языке Java вместо абстрактного класса, подлежащего реализации, можно использовать интерфейс</p> |

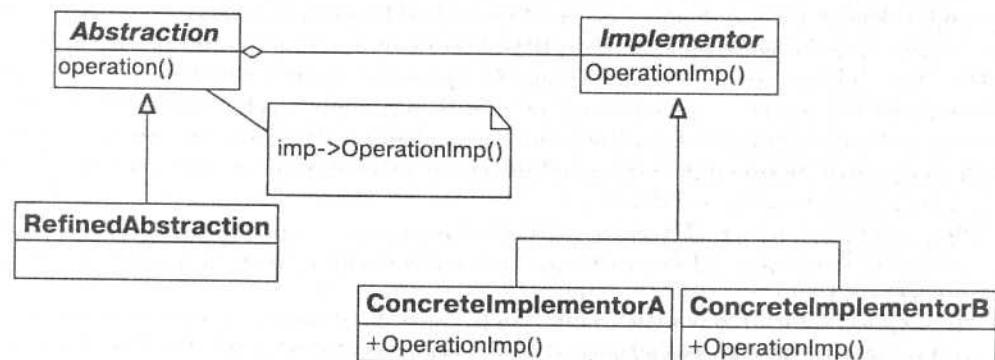


РИС. 9.15. Стандартное упрощенное представление шаблона Bridge

Когда несколько шаблонов интегрируются в одно целое (подобно шаблонам Bridge и Adapter в нашем примере), результатом является так называемый составной шаблон проектирования.⁷ Это позволяет нам говорить о шаблонах, построенных из шаблонов проектирования!

Еще один интересный момент, который следует отметить, заключается в том, что объекты, представляющие абстракцию (*Shape*), получают доступ к функциям реализации уже после их инициации. Эта особенность не является характерной чертой данного шаблона, однако встречается очень часто.

Теперь, когда мы разобрались, что же такое шаблон Bridge, будет полезно вновь обратиться к работе "банды четырех", а именно – к разделу, в котором описывается реализация шаблона. Здесь обсуждаются различные проблемы, касающиеся того, как объекты абстрактной части шаблона создают и/или используют объекты части реализации.

Иногда при применении шаблона Bridge требуется организовать совместное использование объектов части реализации несколькими объектами, принадлежащими различным абстракциям.

- В языке Java это не вызывает никаких проблем, поскольку, когда все объекты стороны абстракции удаляются, программа автоматической сборки мусора устанавливает, что объекты на стороне реализации также больше не нужны, и удаляет их.
- В языке C++ необходимо каким-то образом управлять объектами реализации, для чего существует много способов. Например, можно организовать хранение счетчика ссылок или даже применить шаблон Singleton. Но все же лучше, когда нет необходимости прилагать дополнительные усилия для решения этого вопроса. Это иллюстрирует еще одно важное преимущество автоматической сборки мусора.

Несмотря на то что решение, которое было получено с применением шаблона Bridge, намного превосходит первоначальное, оно все же несовершенно. Оценить качество проекта можно проанализировав, насколько эффективно он позволяет вносить изменения в систему. При использовании шаблона Bridge внесение новых версий реализации существенно упрощается. Программисту достаточно определить новый конкретный класс и реализовать его. Больше никаких изменений не потребуется.

Однако ситуация оказывается существенно хуже, если в систему необходимо включить новый конкретный вариант абстракции. Безусловно, может оказаться, что для отображения нового типа фигуры будет достаточно тех функций реализации, которые уже существуют в системе. Однако это может быть и такая фигура, отображение которой потребует использования новых функций графической программы. Например, может потребоваться организовать в системе поддержку эллипсов. Существующий класс *Drawing* не содержит метода, позволяющего корректно отображать

⁷ Составные (composite) шаблоны проектирования ранее было принято называть сложными (composite), но сейчас предпочтение отдается названию составные, чтобы избежать путаницы с шаблоном Composite.

Подробности – "Riehl, D., Composite Design Patterns, In, Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97), New York: ACM Press, 1997, с. 218–228". Кроме того, представляет интерес публикация "Composite Design Patterns (They Aren't What You Think), C++ Report, June 1998".

эллипсы. В этом случае потребуется внести изменения и на стороне реализации. Однако нам, по крайней мере, понятно, как это происходит — модификация интерфейса класса *Drawing* (или интерфейса *Drawing* в языке Java) и соответствующие модификации всех его производных классов. Таким образом, место внесения изменений точно локализуется, что снижает риск появления нежелательных побочных эффектов.

Подсказка. Использование шаблонов не всегда означает автоматическое получение самых совершенных решений. Однако благодаря тому, что шаблоны представляют собой коллективный опыт многих проектировщиков, накопленный за долгие годы, они часто оказываются лучше тех решений, которые можно было бы быстро придумать самостоятельно.

В реальном мире новые проекты редко должны включать поддержку многовариантной реализации с самого начала. Иногда мы знаем, что появление новых реализаций *возможно*, но возникают они всегда неожиданно. Один из подходов состоит в том, чтобы заранее подготовиться к многовариантной реализации, всегда разделяя систему на стороны абстракции и реализации. В результате получится хорошо структурированное приложение.

Но этот подход не стоит рекомендовать, поскольку он ведет к нежелательному увеличению количества классов в системе. Важно создавать программный код таким способом, чтобы в случае, если многовариантная реализация все же потребуется (а она будет требоваться часто), существовал простой способ модифицировать систему, включив в нее шаблон *Bridge*. Модификация кода для улучшения его структуры без добавления функций называется *рефакторингом* (refactoring). По определению Мартина Фаулера (Martin Fowler), "рефакторинг — это процесс такого изменения программного обеспечения, при котором внешнее поведение кода не изменяется, но его внутренняя структура все же улучшается"⁸.

При проектировании программного кода я всегда уделял внимание возможности его рефакторинга, следуя стратегии "одно правило, одно место". Хорошим примером применения подобного подхода является метод *DrawLine()*. Хотя фактически фрагменты кода размещены в различных местах, перемещаться по нему совсем просто.

Рефакторинг

Рефакторинг часто применяется в объектно-ориентированном проектировании. Однако данное понятие принадлежит не только ООП. Рефакторинг — это просто модификация кода с целью улучшить его структуру без добавления новых функций.

При выведении шаблона мы взяли две изменяемые части проекта (фигуры и графические программы) и инкапсулировали каждую из них в собственный абстрактный класс. Вариации в фигурах — в классе *Shape*, а вариации в графических программах были инкапсулированы в классе *Drawing*.

Посмотрим со стороны на две полиморфные структуры и спросим: "Что представляют данные абстрактные классы?". Очевидно, что класс *Shape* представляет различные виды фигур, а класс *Drawing* — то, как эти фигуры будут отображены. Таким образом, даже если к классу *Drawing* предъявляются новые требования (например, появ-

⁸ Fowler M., Refactoring: Improving the Design of Existing Code, Reading, MA: Addison-Wesley, 2000, c. xvi.

ляется необходимость реализовать поддержку эллипсов), между классами сохраняются ясные отношения.

Резюме

Для того чтобы изучить шаблон Bridge, мы рассмотрели проблему, при которой в предметной области присутствуют две вариации — геометрических фигур и графических программ. В заданной предметной области каждая вариация изменяется независимо. Трудности появились при попытке воспользоваться решением, построенным на учете всех возможных конкретных ситуаций взаимодействия. Подобное решение, примитивным образом использующее механизм наследования, приводит к созданию громоздкого проекта, характеризующегося сильной связанностью и слабой связностью, а следовательно, крайне неудобного в сопровождении.

При обсуждении шаблона Bridge мы следовали следующим стратегиям работы с вариациями.

- Найдите то, что изменяется, и инкапсулируйте это.
- Отдавайте предпочтение композиции, а не наследованию.

Выявление того, что изменяется, — важный этап изучения предметной области. В примере с графическими программами мы столкнулись с тем, что один набор вариаций использует другой. Это и есть показатель того, что в данной ситуации шаблон Bridge может оказаться весьма полезным.

В общем случае, чтобы определить, какой именно шаблон лучше использовать в данной ситуации, следует сопоставить его с характеристиками и поведением существенных в проблемной области. Зная ответы на вопросы *зачем* и *что* в отношении всех известных вам шаблонов, несложно будет выбрать среди них именно те, которые позволяют решить проблему. Шаблоны могут быть выбраны еще до того, как станет известен способ их реализации.

Применение шаблона Bridge позволяет получить более устойчивые проектные решения для представления элементов абстракции и реализации, упрощая их возможное последующее изменение.

Завершая обсуждение шаблона, будем полезным еще раз напомнить о тех принципах объектно-ориентированного проектирования, которые используются в шаблоне Bridge (табл. 9.2).

Таблица 9.2. Принципы ООП, используемые в шаблоне Bridge

| Концепция | Обсуждение |
|---------------------------------------|---|
| Объекты отвечают только за самих себя | Существует несколько классов, представляющих фигуры, но каждый из них отображает сам себя (с помощью метода <code>draw()</code>). Классы Drawing отвечают за отображение элементов этих фигур |
| Абстрактный класс | Абстрактные классы используются для представления концепций. Реально в проблемной области мы имели дело с прямоугольниками и окружностями. Концепция "фигура" — это нечто, существующее только в нашей голове, средство |

Окончание таблицы

| Концепция | Обсуждение |
|--------------------------------------|--|
| Инкапсуляция через абстрактный класс | <p>для связывания двух реалий между собой. Поэтому она была представлена в системе абстрактным классом <i>Shape</i>. Ни один экземпляр объекта класса <i>Shape</i> никогда не будет создан в системе, поскольку он не существует в проблемной области сам по себе — в ней существуют только прямоугольники и окружности. Это же относится и к графическим программам</p> <p>В этой задаче у нас есть два примера инкапсуляции посредством создания абстрактного класса.</p> <ul style="list-style-type: none"> Объект-клиент, обращающийся к шаблону Bridge, всегда будет иметь дело только с объектом класса, производного от класса <i>Shape</i>. Клиент не будет знать, с каким именно из конкретных подтипов класса <i>Shape</i> он взаимодействует — для клиента это всегда будет только класс <i>Shape</i>. Следовательно, имеет место инкапсуляция информации. Преимущество подобного подхода состоит в том, что появление нового типа фигуры в будущем никак не затрагивает объект-клиент. Класс <i>Drawing</i> скрывает от класса <i>Shape</i> наличие в системе различных графических программ. На практике объекты со стороны абстракции смогут узнать, какая именно реализация используется, поскольку именно они создают соответствующие экземпляры объектов. В некоторых случаях это может оказаться полезным. Однако даже в тех случаях, когда это имеет место, знание о конкретной реализации ограничено функцией-конструктором объекта абстракции и, следовательно, легко модифицируемо |
| Одно правило, одно место | Абстрактный класс часто включает методы, которые непосредственно используют объекты из части реализации. Конкретные классы, производные от данного абстрактного, должны вызывать именно эти методы. Такой подход упрощает модификацию системы, когда в этом появляется необходимость, а также дает хорошую отправную точку в проектировании еще до полной реализации шаблона |

Приложение. Примеры программного кода на языке C++

Листинг 9.4. Только прямоугольники

```

void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
}

void V1Rectangle::drawLine
(double x1, double y1,
 double x2, double y2) {
    DP1.draw_a_line(x1, y1, x2, y2);
}

void V2Rectangle::drawLine
(double x1, double y1,
 double x2, double y2) {
    DP2.drawline(x1, x2, y1, y2);
}

```

Листинг 9.5. Прямоугольники и окружности без использования шаблона Bridge

```

class Shape {
    public: void draw () = 0;
}
class Rectangle : Shape {
    public:
        void draw();
    protected:
        void drawLine(
            double x1, y1, x2, y2) = 0;
}
void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
}
// Классы V1Rectangle и V2Rectangle порождены от
// класса Rectangle. Заголовок файла не показан
void V1Rectangle::drawLine (
    double x1, y1, x2, y2) {
    DP1.draw_a_line(x1, y1, x2, y2);
}
void V2Rectangle::drawLine (
    double x1,y1, x2,y2) {
    DP2.drawline(x1,x2,y1,y2);
}

class Circle : Shape {
    public:
        void draw() ;
    protected:
        void drawCircle(
            double x, y, z) ;

```

```

}

void Circle::draw () {
    drawCircle();
}

// Классы V1Circle и V2Circle порождены от
// класса Circle. Заголовок файла не показан
void V1Circle::drawCircle (
    DP1.draw_a_circle(x, y, r);
}

void V2Circle::drawCircle (
    DP2.drawcircle(x, y, r);
}

```

Листинг 9.6. Фрагменты кода — реализация с применением шаблона Bridge

```

void main (String argv[]) {
    Shape *s1;
    Shape *s2;
    Drawing *dp1, *dp2;

    dp1= new V1Drawing;
    s1=new Rectangle(dp, 1, 1, 2, 2);

    dp2= new V2Drawing;
    s2= new Circle(dp, 2, 2, 4);

    s1->draw();
    s2->draw();

    delete s1; delete s2;
    delete dp1; delete dp2;
}

// Замечание. Управление памятью не тестировалось.
// Файлы Include не показаны.

class Shape {
public: draw() = 0;
private: Drawing *_dp;
}
Shape::Shape (Drawing *dp) {
    _dp = dp;
}
void Shape::drawLine(
    double x1, double y1,
    double x2, double y2)
    _dp -> drawLine(x1, y1, x2, y2);
}

Rectangle::Rectangle (Drawing *dp,
    double x1, y1, x2, y2) :
    Shape(dp) {
    _x1 = x1; _x2 = x2;
    _y1 = y1; _y2 = y2;
}
void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
}

```

```
    drawLine(_x1, _y2, _x1, _y1);
}

class Circle {
public: Circle (
    Drawing *dp,
    double x, double y, double r);
};

Circle::Circle (
    Drawing *dp,
    double x, double y,
    double r) : Shape(dp) {
    _x = x;
    _y = y;
    _r = r;
}

Circle::draw () {
    drawCircle(_x, _y, _r);
}

class Drawing {
public: virtual void drawLine (
    double x1, double y1,
    double x2, double y2) = 0;
};

class V1Drawing :
public Drawing {
public: void drawLine (
    double x1, double y1,
    double x2, double y2);
void drawCircle(
    double x, double y, double r);
};

void V1Drawing::drawLine (
    double x1, double y1,
    double x2, double y2) {
    DP1.draw_a_line(x1, y1, x2, y2);
}

void V1Drawing::drawCircle (
    double x1, double y, double r) {
    DP1.draw_a_circle (x, y, r);
}

class V2Drawing : public
Drawing {
public:
void drawLine (
    double x1, double y1,
    double x2, double y2);
void drawCircle(
    double x, double y, double r);
};

void V2Drawing::drawLine (
    double x1, double y1,
    double x2, double y2) {
    DP2.drawline(x1, x2, y1, y2);
}
```

```
void V2Drawing::drawCircle (
    double x, double y, double r) {
    DP2.drawcircle(x, y, r);
}

// Реализация показана для классов DP1 и DP2
class DP1 {
public:
    static void draw_a_line (
        double x1, double y1,
        double x2, double y2);
    static void draw_a_circle (
        double x, double y, double r);
};

class DP2 {
public:
    static void drawline (
        double x1, double x2,
        double y1, double y2);
    static void drawcircle (
        double x, double y, double r);
};
```

Глава 10

Шаблон Abstract Factory

Введение

Продолжим изучение шаблонов и рассмотрим шаблон Abstract Factory (абстрактная фабрика), предназначенный для создания семейств объектов.

В этой главе мы выполним следующее.

- Выведем шаблон Abstract Factory на основании конкретного примера.
- Рассмотрим ключевые особенности этого шаблона.
- Применим шаблон Abstract Factory к решению задачи о различных версиях САПР.

Назначение шаблона проектирования Abstract Factory

"Банда четырех" определяет назначение шаблона Abstract Factory как "предоставление интерфейса для создания семейств связанных между собой или зависимых друг от друга объектов без указания их конкретных классов".¹

Иногда встречаются ситуации, когда требуется координированно создать экземпляры нескольких объектов. Например, для отображения интерфейса пользователя система может использовать два различных набора объектов для работы в двух различных операционных системах. Применение шаблона проектирования Abstract Factory гарантирует, что система всегда реализует именно тот набор объектов, который отвечает данной ситуации.

Описание шаблона проектирования Abstract Factory на примере

Представим себе, что перед нами поставлена задача создать компьютерную систему для отображения на мониторе и вывода на печать геометрических фигур, описание которых хранится в базе данных. Значение уровня разрешения при выводе на печать и отображении фигур зависит от характеристик того компьютера, на котором функционирует система, — например, быстродействия процессора и объема доступной оперативной памяти. Иными словами, система должна самостоятельно определять уровень требований, предъявляемых ею к конкретному компьютеру.

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 87.

Решение данной задачи сводится к тому, что система должна управлять набором используемых драйверов операционной системы. На маломощных компьютерах она должна загружать драйверы, обеспечивающие низкое разрешение, а на более мощных – другие драйверы, обеспечивающие более высокое разрешение (табл. 10.1).

Таблица 10.1. Схема использования драйверов для компьютеров различной мощности

| Тип устройства | Компьютер с низкой производительностью | Компьютер с высокой производительностью |
|----------------|---|---|
| Дисплей | Объект LRDD Драйвер дисплея с низкой разрешающей способностью (Low-Resolution Display Driver) | Объект HRDD Драйвер дисплея с высокой разрешающей способностью (High-Resolution Display Driver) |
| Принтер | Объект LRPD Драйвер принтера с низкой разрешающей способностью (Low-Resolution Print Driver) | Объект HRPD Драйвер принтера с высокой разрешающей способностью (High-Resolution Print Driver) |

В данном примере различные наборы драйверов взаимно исключают друг друга, но это требование не является обязательным. Иногда встречаются ситуации, когда различные семейства будут включать объекты одного и того же класса. Например, на компьютере со средними характеристиками система может использовать драйвер монитора с низким разрешением (LRDD) и, одновременно, драйвер печати с высоким разрешением (HRPD).

Какое семейство объектов использовать в каждом конкретном случае, определяется особенностями проблемной области. В нашем случае концепция объединения объектов в семейства строится на требованиях, которые каждый из объектов предъявляет к системе.

- *Семейство драйверов с низкой разрешающей способностью* – объекты LRDD и LRPD. Это драйверы, предъявляющие низкие требования к оборудованию компьютера.
- *Семейство драйверов с высокой разрешающей способностью* – объекты HRDD и HRPD. Это драйверы, предъявляющие повышенные требования к оборудованию компьютера.

Первая мысль, которая приходит в голову при обдумывании реализации, – использовать для управления выбором драйверов переключатель, как показано в листинге 10.1.

Листинг 10.1. Вариант 1. Использование переключателя для выбора типа драйвера

```
// Фрагмент кода на языке Java
class ApControl {
    .
    .
    void doDraw () {
        .
        .
        switch (RESOLUTION) {
```

```
case LOW:
    // Использование драйвера LRDD
case HIGH:
    // Использование драйвера HRDD
}
}

void doPrint () {
    switch (RESOLUTION) {
        case LOW:
            // Использование драйвера LRPD
        case HIGH:
            // Использование драйвера HRPD
    }
}
```

Конечно, это вполне работоспособное решение, но оно не лишено кое-каких недостатков. Так, правило определения, какой именно драйвер использовать, смешано здесь с собственно использованием этого драйвера. В результате возникают проблемы и со связанностью, и со связностью.

- *Сильная связанность.* Если изменятся правила определения допустимого разрешения (например, необходимо будет ввести средний уровень разрешения), то потребуется изменить программный код в двух местах, чтобы не разрывать существующие связи.
- *Слабая связность.* Методам `doDraw` и `doPrint` приходится дважды передавать соответствующий набор параметров — оба эти метода должны самостоятельно определить, какой именно драйвер следует использовать, а затем отобразить геометрическую фигуру.

Конечно, сильная связанность и слабая связность в настоящее время не представляют большой проблемы, однако они, безусловно, способствуют повышению затрат на сопровождение в будущем. Кроме того, в реальной ситуации мы, вероятнее всего, столкнемся с необходимостью модифицировать код в большем количестве мест, чем показано в этом простом примере.

Еще одна возможная альтернатива состоит в использовании механизма наследования. Мы можем создать два различных класса `ApControl`: один из них будет использовать драйверы с низкой разрешающей способностью, а другой — с высокой. Оба класса будут производными от одного и того же абстрактного класса, что позволит воспользоваться одним и тем же программным кодом. Этот вариант решения представлен на рис. 10.1.



РИС. 10.1. Вариант 2. Реализация вариаций с помощью механизма наследования

Применение наследования может дать хороший эффект в этом простом случае, однако в целом второй вариант решения также имеет много недостатков, как и первый вариант с переключателем.

- **Комбинаторный взрыв.** Для каждого из существующих семейств и для всех новых семейств, которые появятся в будущем, необходимо создавать новый конкретный класс (т.е. новую конкретную версию класса **ApControl**).
- **Неское назначение.** Вновь образованные классы не способны помочь нам разобраться в том, что происходит в системе. Каждый специализированный класс жестко привязан к конкретному случаю. Чтобы обеспечить простоту сопровождения создаваемого программного кода в будущем, необходимо приложить усилия и сделать назначение каждого фрагмента как можно более понятным. Тогда не потребуется тратить много времени на попытки вспомнить, какой участок кода какие функции выполняет.
- **Композиция объектов предпочтительней наследования.** Наконец, выбранный подход нарушает важнейшее правило разработки — предпочтительнее использовать композицию объектов, а не наследование.

Наличие переключателя свидетельствует о необходимости обращения к абстракции

Чаще всего наличие переключателя свидетельствует о необходимости поддержки полиморфного поведения или о неверном размещении реализации обязательств. В любом случае будет полезно подыскать более общее решение — скажем, ввести новый уровень абстракции или передать соответствующие обязательства другим объектам.

Опыт показывает, что наличие в коде переключателя часто указывает на необходимость применения абстракции. В нашем примере объекты LRDD и HRDD являются драйверами для дисплея, а объекты LRPD и HRPD – драйверами для вывода на принтер. Соответствующие абстракции можно назвать *DisplayDriver* (драйвер дисплея) и *PrintDriver* (драйвер принтера). На рис. 10.2 представлено концептуальное решение для такого подхода. Решение следует считать концептуальным, поскольку классы LRDD и HRDD в действительности не являются производными от одного и того же абстрактного класса.

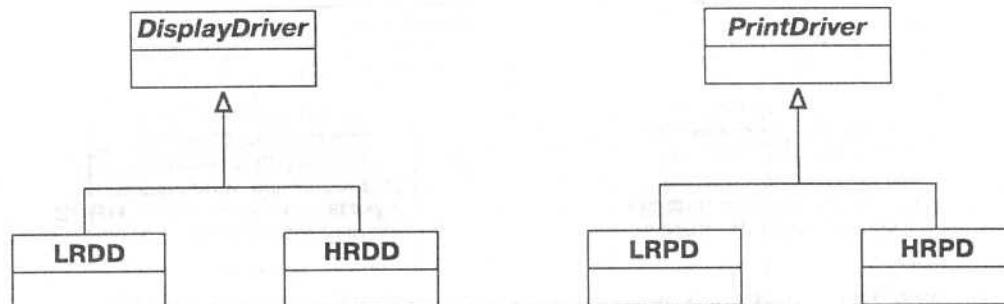


РИС. 10.2. Типы драйверов и соответствующие абстракции

Замечание. На этом этапе мы можем не обращать внимания на то, что классы на схеме являются производными от различных классов, поскольку уже знаем, как можно использовать шаблон Adapter для адаптации драйверов, чтобы создать иллюзию их принадлежности к общему абстрактному классу.

В результате подобного определения объектов класс *ApControl* сможет использовать классы *DisplayDriver* и *PrintDriver* без применения переключателя. Теперь понять работу класса *ApControl* будет намного проще, поскольку отпада необходиомсть анализировать, какой именно тип драйвера он должен использовать. Другими словами, класс *ApControl* может обращаться к классу *DisplayDriver* или классу *PrintDriver*, не заботясь о разрешении, обеспечиваемом драйверами.

Соответствующая схема представлена на рис. 10.3, а программный код на языке Java приведен в листинге 10.2.

Листинг 10.2. Фрагмент программного кода. Решение задачи с помощью методов полиморфизма

```

// Фрагмент кода на языке Java

class ApControl {
    void doDraw () {
        myDisplayDriver.draw();
    }
    void doPrint () {
        myPrintDriver.print();
    }
}
  
```

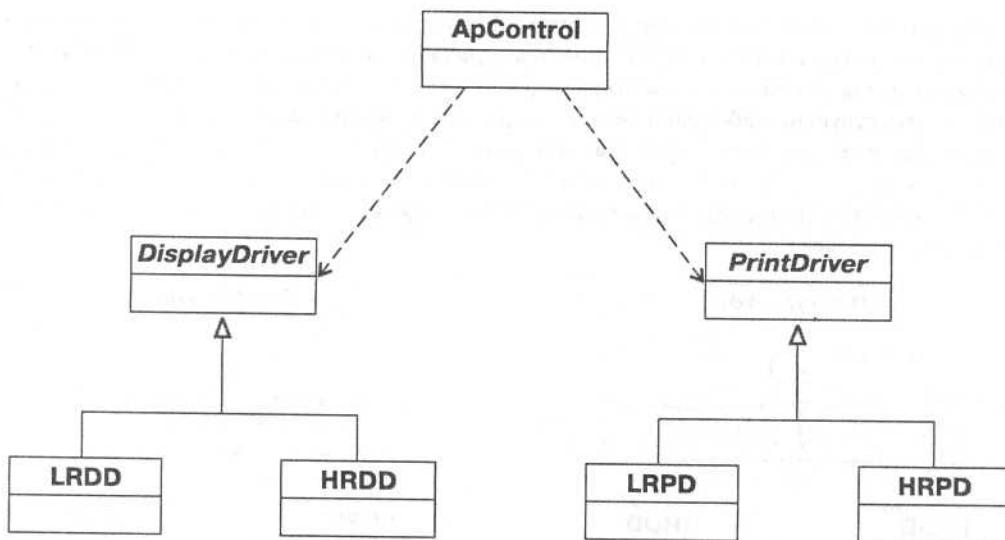


РИС. 10.3. Использование драйверов классом *ApControl* в идеальной ситуации

Остается нерешенным только один вопрос — как создать соответствующие объекты в каждом конкретном случае?

Можно возложить эту задачу на класс **ApControl**, однако подобное решение вызовет проблемы с поддержкой системы в будущем. Если потребуется организовать работу с новым набором объектов, класс **ApControl** неизбежно придется модифицировать. Однако если для создания экземпляров нужных нам объектов использовать некоторый специализированный "объект-фабрику", можно решить проблему, связанную с появлением нового семейства объектов.

В нашем примере объект-фабрика будет использоваться для управления созданием требуемого семейства объектов драйверов. Объект **ApControl** будет обращаться к другому объекту — объекту-фабрике, чтобы создать объекты драйверов дисплея и печати с требуемым разрешением, определяемым в зависимости от характеристик используемого компьютера. Схема подобного взаимодействия представлена на рис. 10.4.

Теперь с точки зрения объекта **ApControl** все выглядит предельно просто. Функции определения, какой именно тип драйвера и как требуется создать, возложена на объект **ResFactory**. Хотя перед нами по-прежнему стоит задача написать программный код, реализующий все необходимые действия по созданию требуемых объектов, мы выполнили декомпозицию проблемы с распределением обязательств между отдельными участниками. Класс **ApControl** должен знать, как работать с соответствующими слушаю объектами, а класс **ResFactory** — решать, какие именно объекты являются соответствующими в каждом конкретном случае. Теперь можно использовать либо различные объекты-фабрики, либо один-единственный объект, содержащий переключатель. В любом случае данный подход будет лучше прежних.

Такое решение вносит в систему необходимую законченность. Назначение объекта класса **ResFactory** состоит лишь в том, чтобы создавать соответствующие объекты драйверов, а назначение объекта класса **ApControl** — в том, чтобы их использовать.

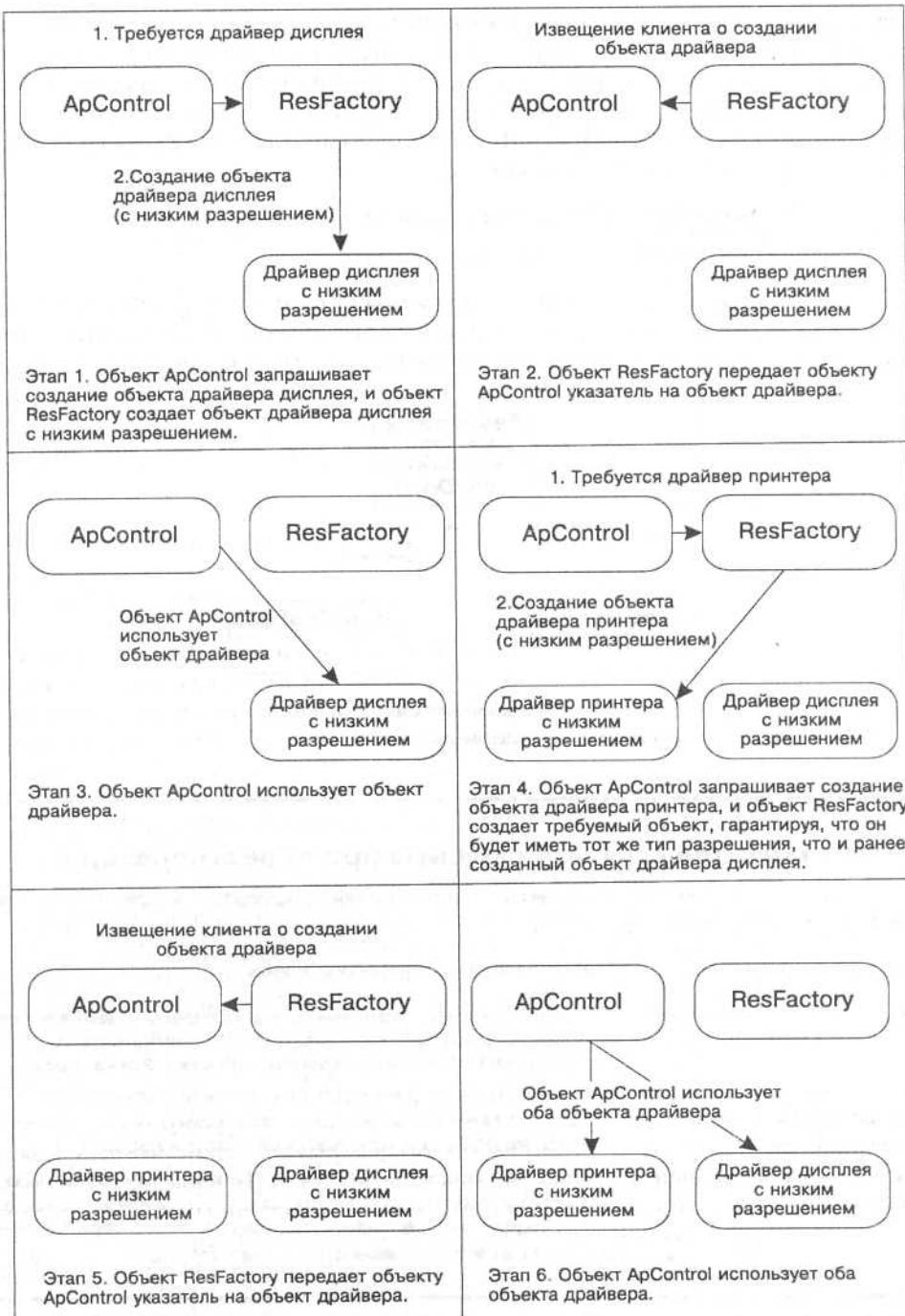


РИС. 10.4. Объект ApControl создает объекты драйверов с помощью объекта-фабрики ResFactory

Существуют различные способы избежать использования переключателя в теле класса **ResFactory**. Их применение позволит вносить возможные изменения, не затрагивая уже существующие объекты-фабрики. Например, можно инкапсулировать возможные вариации за счет определения абстрактного класса, представляющего общую концепцию объекта-фабрики. В нашем примере класс **ResFactory** реализует два различных типа поведения (или метода).

- Создание объекта требуемого драйвера дисплея.
- Создание объекта требуемого драйвера печати.

В системе абстрактный класс **ResFactory** может быть представлен объектом одного из двух конкретных классов, каждый из которых является производным от этого абстрактного класса и имеет требуемые открытые методы — как показано на рис. 10.5.

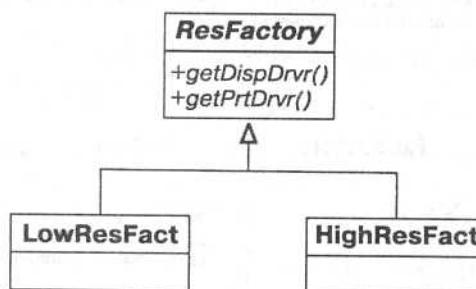


РИС. 10.5. Абстрактный класс **ResFactory** инкапсулирует существующие вариации

Реализация стратегии анализа при проектировании

Ниже описаны три ключевые стратегии проектирования и способы их применения при использовании шаблона *Abstract Factory*.

Стратегия анализа

Найдите то, что изменяется, и инкапсулируйте это

Предпочтительное использование композиции, а не наследования

Проектирование интерфейсов, а не реализаций

Реализация при проектировании

В нашем примере изменчивость заключается в выборе типа объекта драйвера, который требуется использовать. Она была инкапсулирована в абстрактный класс **ResFactory**

Вместо создания двух различных типов объектов **ApControl**, изменчивость вынесена в единственный абстрактный класс **ResFactory**, используемый классом **ApControl**

Класс **ApControl** знает, как потребовать от класса **ResFactory** создать соответствующие экземпляры объектов драйверов, но не знает как именно класс **ResFactory** реализует это требование

Реализация шаблона проектирования Abstract Factory

В листинге 10.3 показано, как объекты шаблона Abstract Factory могут быть реализованы в нашем примере.

Листинг 10.3. Пример реализации объектов класса *ResFactory* на языке Java

```
class LowResFact extends ResFactory {  
  
    DisplayDriver public  
        getDispDrvr() {  
            return new lrdd();  
        }  
  
    PrintDriver public  
        getPrtDrvr() {  
            return new lrpdr();  
        }  
  
}  
class HighResFact extends ResFactory {  
  
    DisplayDriver public  
        getDispDrvr() {  
            return new hrdd();  
        }  
  
    PrintDriver public  
        getPrtDrvr() {  
            return new hrpd();  
        }  
}
```

Для завершения работы над найденным решением необходимо организовать диалог объекта класса **ApControl** с соответствующим объектом-фабрикой (**LowResFact** или **HighResFact**), как показано на рис. 10.6. Обратите внимание на то, что класс **ResFactory** является абстрактным, и именно это сокрытие реализации классом **ResFactory** составляет суть работы данного шаблона. Отсюда и его название — Abstract Factory (абстрактная фабрика).

Объекту класса **ApControl** предоставляется ссылка на объект либо класса **LowResFact**, либо класса **HighResFact**. Он запрашивает создание этого объекта при необходимости получить доступ к соответствующему драйверу. Объект-фабрика создает экземпляр объекта драйвера, самостоятельно определяя его тип (для низкого или высокого разрешения). Причем в этом случае объекту **ApControl** даже нет необходимости знать, какой именно драйвер (с низкой разрешающей способностью или высокой) будет создан, так как со всеми он работает совершенно одинаково.

Мы обошли вниманием один важный вопрос: классы **LRDD** и **HRDD** могут быть производными от разных абстрактных классов (это же замечание может быть справедливо и для классов **LRPD** и **HRPD**). Однако тем, кто знает шаблон Adapter, решить этот вопрос не составит труда. Можно воспользоваться общей схемой, представленной на рис. 10.6, просто добавив к ней элементы адаптации объектов драйверов (рис. 10.7).

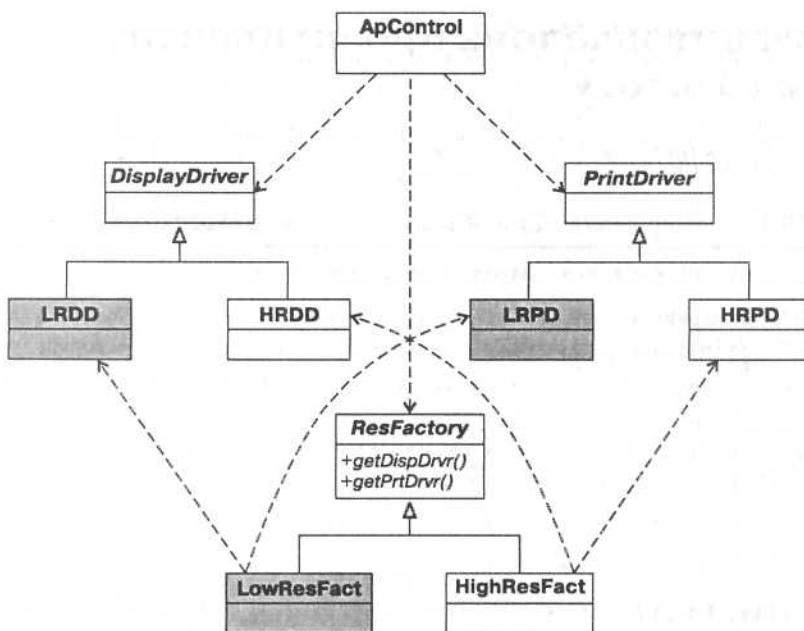


РИС. 10.6. Промежуточное решение с использованием шаблона Abstract Factory

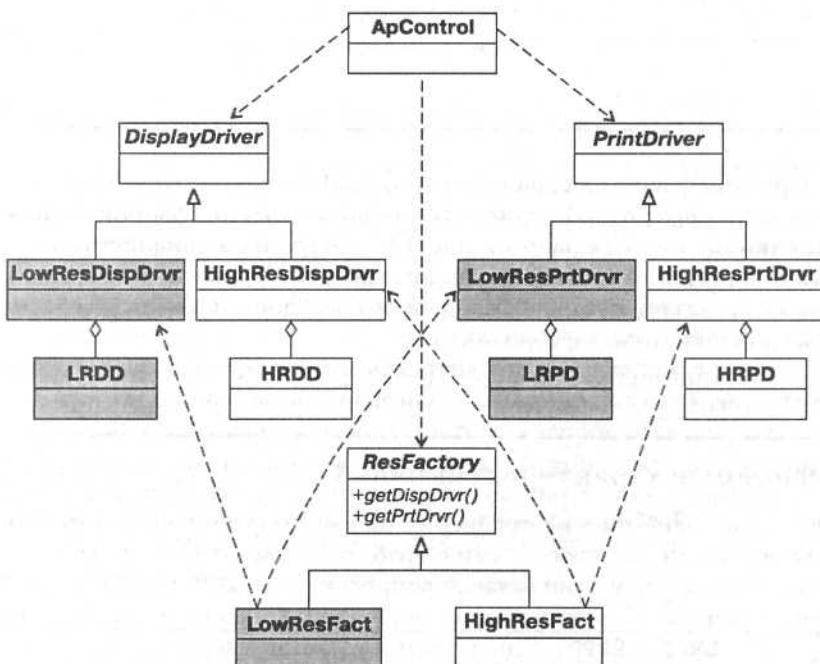


РИС. 10.7. Решение с применением шаблонов Abstract Factory и Adapter

Данная реализация проекта по существу неотличима от прежней. Единственное различие состоит в том, что теперь объекты-фабрики создают экземпляры объектов тех классов, которые были добавлены нами для адаптации объектов предыдущего варианта схемы. Это очень важный метод моделирования. Подобное совместное использование шаблонов Adapter и Abstract Factory позволяет обращаться с концептуально сходными объектами так, как будто они близкородственны друг другу, даже если на самом деле это не так. Такой подход позволяет использовать шаблон Abstract Factory в самых разных ситуациях.

Вот характерные особенности полученного шаблона.

- Клиентский объект знает только, к какому объекту следует обратиться для создания требуемых ему объектов и как их использовать.
- В классе абстрактной фабрики уточняется, экземпляры каких объектов должны быть созданы, для чего определяются отдельные методы для различных типов объектов. Как правило, объект абстрактной фабрики включает отдельный метод для каждого существующего типа объектов.
- Конкретные объекты-фабрики определяют, какие именно объекты должны создаваться.

Дополнительные замечания о шаблоне Abstract Factory

Выбрать, какой именно объект-фабрика должен использоваться, в действительности то же самое, что и определить, какое следует использовать семейство объектов. Например, в обсуждавшейся выше задаче о драйверах мы имели дело с двумя семействами драйверов – одно для систем с низкой разрешающей способностью, а другое для систем с высокой разрешающей способностью. Как узнать, какой набор драйверов следует выбрать? В этом случае, вероятно, потребуется обратиться к файлу конфигурации системы, содержащему описание характеристик компьютера. В программу придется добавить несколько строк программного кода, где на основе полученной информации будет выбираться объект-фабрика с требуемым набором свойств.

Шаблон Abstract Factory также может быть использован совместно с подсистемами различных приложений. В этом случае объект-фабрика будет пропускным пунктом для подсистем, сообщая им, какие именно объекты из их состава должны использоваться. Как правило, главная система знает, какое семейство объектов должно использоваться каждой подсистемой, поэтому до обращения к некоторой подсистеме можно будет создать требуемый экземпляр объекта-фабрики.

Основные характеристики шаблона Abstract Factory

| | |
|----------------|---|
| Назначение | Требуется организовать работу с семействами или наборами объектов для определенных объектов-клиентов (или отдельных случаев) |
| Задача | Необходимо создать экземпляры объектов, принадлежащих заданному семейству |
| Способ решения | Координация процесса создания семейств объектов. Предусматривается выделение реализации правила создания тех или иных семейств объектов из объекта-клиента, который в дальнейшем будет использовать созданные объекты, в отдельный объект |

| | |
|------------|---|
| Участники | Класс <i>AbstractFactory</i> определяет интерфейс для создания каждого из объектов заданного семейства. Как правило, каждое семейство создается с помощью собственного конкретного уникального класса <i>ConcreteFactory</i> |
| Следствия | Шаблон отделяет правила, какие объекты нужно использовать, от правил, как эти объекты следует использовать |
| Реализация | Определяется абстрактный класс, описывающий, какие объекты должны быть созданы. Затем реализуется по одному конкретному классу для каждого семейства объектов (рис. 10.8). Для решения этой задачи также могут использоваться таблицы базы данных или файлы |

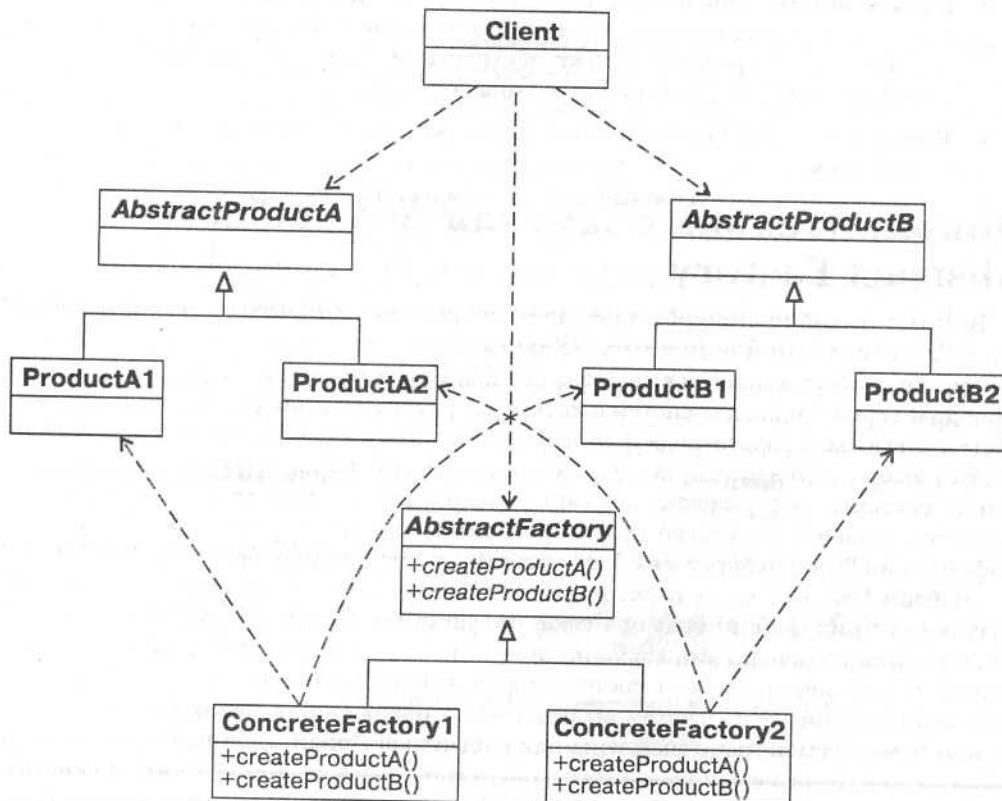


РИС. 10.8. Стандартное упрощенное представление шаблона *Abstract Factory*

На рис. 10.8 показано, что объект **Client** использует объекты, производные от двух различных серверных классов (**AbstractProductA** и **AbstractProductB**). Такой подход упрощает структуру системы, скрывая реализацию, чем упрощает ее сопровождение.

- Объект-клиент не знает, с какой конкретной реализацией серверного объекта он имеет дело, так как ответственность за создание серверного объекта несет объект-фабрика.
- Объект-клиент не знает даже того, к какому конкретному объекту-фабрике он обращается, поскольку он взаимодействует с абстрактным классом **AbstractFactory**, который может быть представлен объектом конкретных классов **ConcreteFactory1** или **ConcreteFactory2**, но каким именно, объект-клиент знать не может.

От объекта класса **Client** полностью скрыта (инкапсулирована) информация о том, какой из серверных объектов используется. Это позволяет в будущем легко вносить изменения в процедуру совершения выбора, так как объект-клиент остается не затронутым.

Шаблон **Abstract Factory** ярко демонстрирует новый вид декомпозиции — декомпозицию ответственности. Подобный подход позволяет разделить задачу на две функциональные стороны.

- Первая сторона, использующая конкретные объекты (класс **ApControl**).
- Вторая сторона, принимающая решение о том, какие именно объекты следует использовать (класс **AbstractFactory**).

Применение шаблона **Abstract Factory** целесообразно в тех случаях, когда проблемная область включает несколько семейств объектов, причем каждое из семейств используется при различных обстоятельствах.

Семейства могут быть определены на основе одного или нескольких критерииев.

Приведем несколько примеров.

- Различные операционные системы (при создании многоплатформенных приложений).
- Различные требования к производительности.
- Различные версии приложений.
- Различные категории пользователей приложения.

Как только семейства и их члены будут идентифицированы, следует принять решение о методе реализации каждого конкретного случая (т.е. каждого семейства). В нашем примере для этой цели определен абстрактный класс, на который возложена ответственность за выбор конкретного типа семейства классов, для которых и будут созданы экземпляры объектов. Затем для каждого семейства от этого абстрактного класса был определен конкретный производный класс, реализующий создание экземпляров объектов для классов — членов семейства.

Иногда встречаются ситуации, когда семейства объектов существуют, но нет необходимости управлять созданием их экземпляров с помощью классов, порожденных специально для каждого семейства. Возможно, более подходящим окажется одно из следующих более динамичных решений.

- Можно использовать файл конфигурации, в котором будет указано, какие объекты следует использовать в каждом случае. Для создания экземпляров требуемых объектов в программе организуется переключатель, управляемый информацией из файла конфигурации.
- Для каждого семейства объектов создается отдельная запись в базе данных, содержащая сведения об используемых объектах. При этом каждый элемент (поле) записи базы данных указывает, какой конкретный класс должен использоваться при вызове соответствующего метода абстрактного класса-фабрики.

При работе с языком Java концепцию использования файла конфигурации можно дополнительно расширить. Имена полей записи файла могут представлять собой имя требуемого класса. В этом случае не требуется даже иметь полное имя класса, достаточно только добавить суффикс или префикс к имени, хранящемуся в файле конфигурации, а затем, используя класс *Class* языка Java, корректно создать экземпляр объекта с данным именем.²

В реально существующих проектах члены различных семейств не всегда являются производными от общего родительского класса. Например, в предыдущем примере с драйверами, возможно, что классы драйверов *LRDD* и *HRDD* будут производными от разных родительских классов. В таких случаях следует так адаптировать эти классы, чтобы шаблон *Abstract Factory* смог работать с ними.

Применение шаблона *Abstract Factory* к проблеме САПР

В задаче с различными САПР система должна работать с различными наборами элементов, определяемыми в зависимости от используемой версии САПР. При работе с САПР версии V1 все элементы должны быть реализованы для версии V1, а при работе с САПР версии V2 все элементы должны быть реализованы для версии V2.

Исходя из этого требования, несложно выделить два семейства классов, для управления которыми следует применить шаблон *Abstract Factory*. Первое семейство – это классы элементов версии V1, а второе – классы элементов версии V2.

Резюме

Шаблон *Abstract Factory* применяется, когда необходимо координировать создание семейств объектов. Он позволяет отделить реализацию правил создания экземпляров новых объектов от объекта-клиента, который будет эти объекты использовать.

- Прежде всего, идентифицируйте правила создания экземпляров объектов и определите абстрактный класс с интерфейсом, включающим отдельный метод для каждого из классов, экземпляры которых должны быть созданы.
- Затем для каждого семейства создайте конкретные классы, производные от данного абстрактного класса.
- Реализуйте в объекте-клиенте обращение к абстрактному объекту-фабрике с целью создания требуемых серверных объектов.

² Подробное описание класса *Class* языка Java можно найти в Eckel B. Thinking in Java, Upper Saddle River, N.J.: Prentice Hall, 2000.

Приложение. Примеры программного кода на языке C++

Листинг 10.4. Фрагмент кода. Переключатель управления выбором драйвера

```
// Фрагмент кода на языке C++
// Класс ApControl
void ApControl::doDraw () {
    .
    .
    switch (RESOLUTION) {
        case LOW:
            // Использование LRDD
        case HIGH:
            // Использование HRDD
    }
}
void ApControl::doPrint () {
    .
    .
    switch (RESOLUTION) {
        case LOW:
            // Использование LRPD
        case HIGH:
            // Использование HRPD
    }
}
```

Листинг 10.5. Фрагмент кода. Использование полиморфизма для решения проблемы

```
// Фрагмент кода на языке C++
// Класс ApControl
void ApControl::doDraw () {
    .
    .
    myDisplayDriver->draw();
}
void ApControl::doPrint () {
    .
    .
    myPrintDriver->print();
}
```

Листинг 10.6. Фрагмент кода. Реализация класса ResFactory

```
// Фрагмент кода на языке C++
class LowResFact : public ResFactory {
    DisplayDriver *
    LowResFact::getDispDrvr() {
        return new lrdd;
    }
}
```

```
PrintDriver *
LowResFact::getPrtDrvrv() {
    return new lrpd;
}

class HighResFact : public ResFactory;

DisplayDriver *
HighResFact::getDispDrvrv() {
    return new hrdd;
}

PrintDriver *
HighResFact::getPrtDrvrv() {
    return new hrpd;
}
```

Глава 11

Как проектируют эксперты

Введение

С чего следует начинать разработку проекта? Определить все возможные детали, а затем искать способ объединить их в единое целое? Или лучше предварительно составить общее представление о проекте и лишь затем приступить к его детализации? А может существует и другой путь?

Подход, предложенный Кристофером Александером, состоит в том, чтобы сначала сосредоточить внимание на отношениях концептуального уровня, а затем продвигаться от общего к частному. Прежде чем принять какое-либо проектное решение, он предлагает глубоко вникнуть в контекст той проблемы, которую это решение должно преодолеть, используя шаблоны проектирования для выявления существующих в этой среде отношений. Однако Александр предлагает не просто коллекцию шаблонов, а самостоятельную методологию проектирования. Предметной областью, о которой он пишет, является архитектура. Ее задача состоит в проектировании зданий, в которых люди живут и работают. Но предложенные Александром принципы вполне применимы и к проектированию программного обеспечения.

В этой главе мы выполним следующее.

- Обсудим подход Александера к проектированию.
- Проанализируем, как применить этот подход к разработке программного обеспечения.

Проектирование посредством добавления различий

После ознакомления с несколькими шаблонами проектирования пришло время выяснить, как они могут работать совместно. Александр полагает, что недостаточно просто описать отдельные шаблоны. Он использует их для построения новой парадигмы проектирования.

Книга Александера *The Timeless Way of Building* посвящена одновременно и описанию шаблонов, и обсуждению методов их совместного использования. Это превосходная книга, одна из моих самых любимых как с личной, так и с профессиональной точки зрения. Она помогла мне осознать то, что происходит в моей жизни, лучше понять тот мир, в котором я живу, а также научиться создавать более качественное программное обеспечение.

Как это стало возможным? Как может книга, посвященная проектированию зданий и городов, оказать столь глубокое влияние на разработку программного обеспечения? Причина, как я полагаю, состоит в том, что в ней Александр представил новую парадигму, которая будет полезна *любому* проектировщику. Именно эта новая парадигма проектирования и является наиболее важным и интересным аспектом его книги.

К сожалению, я не могу похвастать тем, что принял предложенную Александром идеологию после первого же прочтения его книги. Первоначальной моей реакцией была такая мысль: "Это очень интересно. Возможно, все это даже имеет практический смысл". А затем я вновь возвратился к традиционным методам проектирования, которые использовал на протяжении многих лет.

Но иногда старые пословицы все же находят себе подтверждение в жизни. Не даром говорят: "Удача приходит к тем, кто ее ждет" или "На ловца и зверь бежит". Вскоре мне представился подходящий случай, благодаря которому все стало на свои места.

Спустя несколько недель после прочтения книги Александера жизнь заставила меня вновь вернуться к ней. Я был занят в проекте, который не поддавался традиционным подходам — они попросту не работали. Конечно, я мог бы предложить кое-какие решения, но все они были недостаточно хороши. Все мои старые и испытанные методы проектирования терпели неудачу, и я был очень расстроен. К счастью, мне хватило мудрости попробовать новый путь, предложенный Александром, и очень скоро можно было лишь восхищаться полученными результатами.

В следующей главе мы подробно обсудим все, что мной тогда было сделано. Но сначала посмотрим, какой же подход предлагает Александр.

Проектирование часто представляют себе как процесс синтеза, процесс соединения элементов в единое целое, процесс комбинирования. Согласно этому представлению целое создается соединением отдельных частей, т.е. элементы являются первичными, а общая форма — вторичной.¹

Это вполне естественно — проектировать, переходя от частного к общему, начиная с конкретных вещей, понятных и знакомых.

Когда я впервые прочитал это, то подумал: "Все это вполне соответствует моему взгляду на вещи. Сначала выяснить, что мне может пригодиться, а затем собрать все это в одно целое." Иначе говоря, при проектировании сначала идентифицируются требуемые классы, а затем уже устанавливается порядок их взаимодействия. После того как все требуемые элементы будут собраны в одно целое, можно вернуться на шаг назад и оценить, что же у нас получилось. Но и в этом случае при переключении внимания от частного (локального) к общему (глобальному) общая картина остается в нашем представлении состоящей из отдельных частей.

В объектно-ориентированной разработке элементами являются объекты и классы. Именно они идентифицируются на первом этапе, после чего определяется их поведение и интерфейсы. Однако, начав проектирование с частностей, мы, как правило, так и остаемся сосредоточенными на них до конца.

Вспомним исходный вариант решения проблемы с различными версиями САПР, предложенный в главе 4, *Стандартное объектно-ориентированное решение*. Мы начали с рассуждений о различных классах, которые могут нам потребоваться: классы для

¹ Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Building. New York: Oxford University Press, 1979, с. 368.

представления пазов, отверстий, просечек и т.д. Поскольку необходимо было связать их с системами версий V1 и V2, предполагалось наличие двух наборов классов, специализированных для каждой из систем. И только после определения всех требуемых классов внимание было перенесено на проблему их взаимодействия.

Тем не менее, простым соединением заранее определенных частей невозможно сформировать что-либо, имеющее естественный, законченный облик.²

Тезис Александера состоит в том, что построение целого из отдельных частей – это далеко не лучший метод проектирования.

Несмотря на то что речь в книге Александера идет об архитектуре, многие признанные авторитеты в области разработки программного обеспечения согласятся с тем, что этот тезис верен и в их области. Я попытался понять, в чем состоит суть нового подхода к разработке. В результате в моей голове сложилась фраза, как будто бы произнесенная Александером: "Хорошее программное обеспечение не может быть создано простым соединением заранее определенных частей" (т.е. без предварительной оценки, насколько хорошо эти части будут подходить друг другу).

Когда элементы представляют собой отдельные модули, созданные прежде общего целого, то по определению они всегда будут идентичными, и нельзя ожидать, что каждая часть будет уникальной, отвечающей своему положению в общем целом. И что еще более важно, с помощью любой комбинации модульных элементов просто невозможно получить все то множество шаблонов, которые должны быть одновременно представлены в любом месте, предназначенном для пребывания человека.³

Размышления Александера, связанные с модульностью, поначалу были для меня совершенно непонятны. Но в конце концов мне стало ясно, что если начать с создания модулей еще до уяснения того, как будет выглядеть общая картина, то модули в дальнейшем всегда будут одними и теми же, поскольку не существует никаких причин для их изменения в дальнейшем.

Похоже, именно в этом и состоит основная цель концепции многократного использования. Разве она не предполагает постоянного использования одинаковых модулей? Именно так. Однако нам также требуется максимальная гибкость и устойчивость системы. Само по себе простое создание модулей не гарантирует этого.

Ознакомившись с использованием шаблонов проектирования по методологии Александера, я смог создавать многократно используемые и одновременно гибкие классы с гораздо большим успехом, чем прежде. Как проектировщик, я стал намного лучше.

Единственно возможный способ создать жизненное пространство, полностью отвечающее нуждам его обитателей, состоит в максимальном приспособлении каждой части общего целого к той позиции, которую она в нем занимает.⁴

В нашем случае выражение *полностью отвечающее нуждам* следует понимать как *устойчивое и гибкое программное обеспечение*.

Выше упоминались слова Александера о том, что части должны быть уникальны – только в этом случае удастся реализовать все преимущества, определяемые их конкрет-

² Там же, с. 368.

³ Там же, с. 368, 369.

⁴ Там же, с. 369.

ным местоположением. Рассмотрим это утверждение подробнее. Каждый элемент создается посредством копирования эталона с последующим приспособлением новой копии к существующему окружению, что и придает общему целому уникальный, присущий только ему характер. Рассмотрим несколько примеров из области архитектуры.

- *Швейцарская деревня.* Перед глазами встает деревня, состоящая из расположенных близко друг к другу уютных коттеджей, очень похожих один на другой, но, тем не менее, каждый из них имеет что-то индивидуальное. Различия между домами вовсе не произвольны – каждый коттедж отражает финансовые возможности своего создателя и владельца и непременно удовлетворяет требованию гармонично вписаться в окружающую среду. Достигнутый эффект очень хорош – деревня в целом производит впечатление уюта и комфортабельности.
- *Американский пригород.* Все коттеджи выглядят как пряничные домики. Внимание к естественному окружению построек уделяется крайне редко. Установившиеся понятия и стандарты всячески поощряют подобное однообразие. В результате возникает эффект полного обезличивания зданий, и общая картина не способна вызывать каких-либо приятных эмоций.

Безусловно, в данный момент применение обсуждаемого подхода к проектированию программного обеспечения может показаться слишком уж концептуальным. Однако сейчас достаточно понять, что для создания устойчивых и гибких программных систем необходимо разрабатывать их элементы (классы или объекты) с учетом того окружения (контекста) в котором они будут функционировать.

Короче говоря, каждая часть обретает свою специфическую форму за счет ее нахождения в конкретном контексте более общего целого.

Это есть процесс дифференциации. Проектирование рассматривается как серия этапов *сложнения*. Простая структура превращается в нечто более общее, контролируемая и направляемая этим общим, а не за счет простого объединения мелких частей друг с другом. В процессе дифференциации целое порождает свои части, при этом формирование целого и его частей происходит одновременно. В целом процесс дифференциации напоминает развитие эмбриона.⁵

Усложнение – что означает здесь это слово? Ведь наша цель состоит в том, чтобы сделать процесс проектирования проще, а не сложнее!

Смысль его в том, что согласно подходу Александера обдумывание проекта должно начинаться с описания проблемы в простейших терминах и понятиях с последующим добавлением дополнительных особенностей (различий), в результате чего проект будет становиться все более и более сложным благодаря постоянному накоплению анализируемой информации.

Это совершенно естественный процесс. Мы постоянно применяем его на практике. Например, представим, что требуется договориться об аудитории для проведения лекции, причем ожидается, что слушателей будет около 40 человек. Описание соответствующих требований, скорее всего, будет выглядеть примерно так: "Мне понадобится комната размером 10 на 10 метров" (начинаем с простого). Далее: "Потребуются также стулья – 5 рядов по 8 стульев, установленные в виде полукруга" (добавление

⁵ Там же, с. 370.

информации делает описание комнаты более сложным). Наконец: "Необходима также кафедра для лектора, расположенная перед слушателями" (еще более сложное описание).

Развертывание проекта в сознании его создателя, в терминах используемого им языка – это тот же самый процесс.

Каждый шаблон – это оператор, дифференцирующий пространство: т.е. он создает различия там, где прежде их не было. В языке операторы упорядочиваются в последовательности: в соответствии с тем, как они выполняются, один за другим. В результате рождается законченная форма, общая в том смысле, что она имеет структуру, одинаковую с другими сравнимыми элементами, и специфическая в том смысле, что она уникальна в соответствии с ее собственными обстоятельствами.

Язык представляет собой последовательность таких операторов, в которой каждый оператор вносит дальнейшую дифференциацию в образ, являющийся порождением предыдущих дифференциаций.⁶

Итак, Александр утверждает, что проектирование должно начинаться с простейшей формулировки проблемы с последующей постепенной ее детализацией (усложнением) посредством добавления в эту формулировку новой информации. Вносимая информация принимает форму шаблона, поскольку, считает Александр, шаблоны определяют отношения между сущностями в проблемной области.

Для примера еще раз обратимся к шаблону "Внутренний двор", обсуждавшемуся в главе 5, *Первое знакомство с шаблонами проектирования*. Шаблон должен описывать сущности, присутствующие в среде внутреннего двора, и их взаимоотношения между собой. Такими сущностями являются следующие.

- Открытое пространство внутреннего двора.
- Пересекающиеся пути между окружающими помещениями.
- Внешний вид из внутреннего двора.
- И даже люди, которые будут пользоваться этим внутренним двором.

Осмысление проблемы в терминах того, как эти объекты должны взаимодействовать между собой, дает достаточно информации для проектирования внутреннего двора. Затем предварительный проект уточняется за счет применения других шаблонов, которые могут присутствовать в контексте шаблона "Внутренний двор", – например, крыльца или веранды, выходящих во внутренний двор.

Что же делает данный аналитический метод столь мощным, что он не нуждается в подтверждении моим личным опытом, интуицией и творческим потенциалом? Александр утверждает, что существование подобных шаблонов есть объективная реальность, и они существуют независимо от отдельных личностей. Организация пространства отвечает запросам его обитателей тогда, когда она удовлетворяет их естественным потребностям, а не просто потому, что автором проекта является гений. Если качество проекта зависит от его соответствия естественным процессам, не должно вызывать удивления, что качественные решения сходных проблем будут выглядеть очень похоже.

⁶ Там же, с. 372, 373.

Основываясь на этих рассуждениях, Александр сформулировал следующие правила хорошего проектирования.

- *Строго по одному.* Шаблоны должны применяться только по одному, последовательно друг за другом.
- *Сначала следует формировать контекст.* Сначала следует применять те шаблоны, которые создают контекст для других шаблонов.

Шаблоны определяют отношения

Шаблоны, описываемые Александром, определяют отношения между сущностями в проблемной области. Для нас важны не сами шаблоны, а именно эти отношения, поскольку шаблоны просто предлагают способ их анализа.

Подход, предложенный Александром, вполне можно применить и к проектированию программного обеспечения. Конечно, не буквально, а концептуально. Что мог бы сказать Александр, обращаясь к разработчикам программного обеспечения? Возможный набор рекомендаций я поместил в табл. 11.1.

Таблица 11.1. Применение методологии Александера к разработке программного обеспечения

| Последовательность разработки | Пояснения |
|--|---|
| Идентифицируйте шаблоны | Идентифицируйте шаблоны, присутствующие в контексте проблемы. Осмыслите проблему в терминах присутствующих в ней шаблонов. Помните, что назначение шаблонов состоит в определении отношений между сущностями в проблемной области |
| Начните с шаблонов, определяющих контекст | Идентифицируйте те шаблоны, которые создают контекст для других шаблонов. Именно с них следует начать разработку системы |
| Двигайтесь в направлении углубления в контекст | Рассмотрите остальные шаблоны и отыщите любые другие шаблоны, которые ранее остались незамеченными. Из полученного набора опять выделите те шаблоны, которые определяют контекст для оставшихся. Повторяйте эту процедуру до исчерпания всего набора шаблонов |
| Оптимизируйте проект | На этом этапе тщательно проанализируйте общий контекст, полученный в результате применения шаблонов |
| Реализуйте полученную схему | Реализация предусматривает воплощение всех деталей, требуемых каждым из использованных шаблонов |

Личные впечатления автора от использования идей Александера при проектировании программного обеспечения

При первом использовании подхода Александера я воспринял его слова слишком буквально. Его концепции, сформулированные в отношении архитектуры, не всегда удается прямо применить к проектированию программного обеспечения (или другим видам проектирования). В некотором смысле при первых опытах применения шаблонов мне просто повезло, поскольку те проблемы, которые я решал, включали шаблоны проектирования, строго упорядоченные в отношении создания контекста. Однако это везение сработало и против меня, так как по наивности я решил, что так будет всегда (жизнь показала обратное).

Известно, что многие ведущие специалисты в области программного обеспечения поддерживали идею разработки "языка шаблонов" и пытались найти формальный способ применения идей Александера к разработке программного обеспечения. В моем понимании это был бы такой инструмент, который позволил нам прямо применять методы Александера к разработке программного обеспечения (лично я больше не верю, что это возможно). Поскольку Александр указал, что в архитектуре шаблоны обладают заранее определенным порядком образования контекста, я полагал, что и в программном обеспечении шаблоны также имеют некоторый предопределенный порядок. Иначе говоря, один тип шаблона всегда будет создавать контекст для шаблона другого типа. Я начал пропагандировать подход Александера именно так, как я понимал его тогда, и даже учить этому других. Но по прошествии нескольких месяцев и после применения этого подхода в нескольких проектах я столкнулся с серьезными проблемами. Обнаружились ситуации, в которых заранее установленный порядок контекстов не срабатывал.

Как человеку, имеющему математическое образование, мне было достаточно единственного исключения из правил для опровергения всей выдвинутой мной теории. Сложившаяся ситуация заставила меня заново критически пересмотреть весь подход к идеологии шаблонов проектирования. Ранее я всегда придерживался этого правила, но на этот раз просто забыл о нем в радостном возбуждении.

Начиная с самого первого этапа, я вновь проанализировал те *принципы*, на которых строится метод Александера. Несмотря на то, что они по-разному проявляются в архитектуре и в проектировании программ, эти принципы все же полностью применимы к разработке программного обеспечения. Они позволяют повысить качество создаваемых проектов, ускорить их разработку и выполнить более полный анализ. Подтверждение этому я вижу каждый раз при необходимости внесения очередных изменений в созданное мной программное обеспечение.

Резюме

Проектирование обычно понимается как процесс синтеза – процесс объединения отдельных частей в единое целое. При создании программного обеспечения обычный подход состоит в том, чтобы сначала выделить все необходимые объекты, классы и компоненты, и лишь затем подумать о том, как они будут взаимодействовать.

В книге *The Timeless Way of Building* Кристофер Александер предложил лучший подход, основанный на применении шаблонов проектирования. Этот подход предлагает следующее.

1. Начать с концептуального описания всей проблемы в целом, чтобы понять, какие требования в конечном счете должны быть удовлетворены.
2. Идентифицировать шаблоны, которые присутствуют в концептуальном описании проблемы.
3. Начать работу с тех шаблонов, которые создают контекст для остальных.
4. Применить эти шаблоны.

5. Повторить эту процедуру с оставшимися шаблонами, а также с любыми новыми шаблонами, которые, возможно, будут выявлены в процессе проектирования.
6. Наконец, оптимизировать проект и его реализацию в контексте, созданном за счет последовательного применения всех выявленных шаблонов проектирования.

Разработчик программного обеспечения не всегда имеет возможность применить предложенный Александром подход непосредственно. Однако проектирование методом добавления концепций в пределах контекста, образованного за счет представления предыдущих концепций, несомненно, доступно каждому. Помните об этом при изучении новых шаблонов в последующих главах книги. Многие шаблоны позволяют создавать надежное программное обеспечение, потому что они определяют контексты, в пределах которых классы, реализующие некоторое решение, могут успешно работать.

Глава 13

Обработка возможных вариаций с помощью шаблонов проектирования

Введение

В предыдущих главах было показано, каким образом шаблоны проектирования могут применяться как на локальном, так и на глобальном уровнях. На локальном уровне они демонстрируют, как решить определенную проблему в рамках контекста соответствующего шаблона. На глобальном уровне шаблоны представляют схему взаимосвязей компонентов приложения. Один из способов освоения шаблонов проектирования состоит в изучении наиболее эффективных методов их использования как на глобальном, так и на локальном уровнях, которые в совокупности представляют собой прекрасный инструмент решения проблем.

Другой способ изучения шаблонов проектирования заключается в ознакомлении с теми закономерностями, принципами и алгоритмами, которые положены в их основу. Полученные знания помогут вам существенно развить свои способности как аналитика и проектировщика. Изучение этих принципов и алгоритмов поможет найти выход даже в тех ситуациях, когда требуемые шаблоны проектирования еще не созданы, поскольку набор строительных блоков, необходимых для решения проблемы, уже будет вам известен.

В этой главе мы выполним следующее.

- Познакомимся с принципом открытости-закрытости, положенным в основу многих шаблонов проектирования.
- Обсудим принцип проектирования от контекста, который является важнейшим звеном идеологии шаблонов Александера.
- Рассмотрим принцип включения вариаций.

Принцип открытости-закрытости

Очевидно, что программное обеспечение должно быть расширяемым. Однако всякое изменение программного обеспечения связано с риском внесения ошибок. Для устранения этой дилеммы Берtrand Мейер (Bertrand Meyer) предложил принцип открытости-закрытости¹. Упрощенно данный принцип можно сформулировать так:

¹ Meyer B. Object-Oriented Software Construction, Upper Saddle River, N.J.: Prentice Hall, 1997, с. 57.

"Модули, методы и классы должны быть открыты для расширения, но закрыты для модификации".² Другими словами, программное обеспечение следует проектировать таким образом, чтобы можно было расширять его возможности, не изменяя то, что уже существует.

Как бы противоречиво это не звучало поначалу, мы, тем не менее, уже встречались с подобными примерами. При обсуждении шаблона Bridge была продемонстрирована возможность добавления новых реализаций (т.е. расширения возможностей программного обеспечения) без модификации какого-либо из существующих классов.

Принцип проектирования от контекста

Александер рекомендует проектировать, отталкиваясь от контекста, сначала создавая общую картину, а потом переходя к проектированию деталей образующих ее элементов. Большинство шаблонов следуют именно такому подходу, одни в большей степени, другие в меньшей. Из тех четырех шаблонов, с которыми мы уже познакомились, шаблон Bridge является наиболее ярким примером применения данного правила.

Взгляните еще раз на схему шаблона Bridge, приведенную в главе 9, *Шаблон Bridge* (рис. 9.13). Принимая решение о том, как проектировать классы на стороне реализации, учитывайте их контекст – т.е. каким образом классы, производные от класса **Abstraction**, будут их использовать.

Например, если разрабатывается система, предназначенная для вывода изображения разнообразных геометрических фигур на оборудование различного типа, ей обязательно потребуется несколько различных типов классов реализаций, для чего целесообразно использовать шаблон Bridge. Общая схема шаблона Bridge свидетельствует, что классы геометрических фигур будут использовать классы реализации (т.е. различные версии графических программ) через общий интерфейс. В данном случае проектирование от контекста, как рекомендует Александр, означает, что сначала следует рассмотреть, что представляют собой фигуры, которые необходимо изобразить – т.е. выяснить, что, собственно, предстоит отображать. Именно эти требования будут определять поведение классов стороны реализации. Так, от этих классов (графических программ), безусловно, потребуется способность отображать линии, окружности и т.д.

Применение методов анализа общности/изменчивости в приложении к тому контексту, в пределах которого существуют интересующие нас классы, позволяет выявить различные случаи их использования (прецеденты), как существующие, так и потенциальные. В результате можно будет принять обоснованное решение о желательном уровне обобщения (генерализации) на стороне реализации, основываясь на предполагаемых затратах, которых потребует поддержка того или иного уровня обобщения. Такой подход часто позволяет найти более общее решение для стороны реализации, чем это ожидалось поначалу, при этом требующее минимальных дополнительных издержек.

Например, для отображения геометрических фигур, на первый взгляд, вполне достаточно линий и окружностей. Однако давайте зададим себе вопрос: "Отображение каких фигур не может быть выполнено с помощью только прямых линий и дуг окружностей?". Ответ очень прост – это эллипсы. Теперь нам потребуется сделать выбор между следующими вариантами.

² Прекрасную статью "The Open-Closed Principle" Роберта Мартина (Robert C. Martin) можно найти на Web-странице <http://www.netobjectives.com/dpexplained>.

- Дополнительно к линиям и окружностям реализовать отображение эллипсов.
- Учитывая, что эллипс является обобщением понятия окружности, реализовать отображение эллипсов вместо окружностей.
- Отказаться от реализации отображения эллипсов, если связанные с этим издержки не компенсируются полученными преимуществами.

Приведенный выше пример иллюстрирует еще одну важную концепцию проектирования — наличие возможности реализовать что-либо вовсе не означает, что это обязательно должно быть выполнено. Мой опыт работы с шаблонами проектирования показывает, что они позволяют очень хорошо изучить характеристики проблемной области. Однако я крайне редко учитываю все обнаруженные ситуации и чаще всего отказываюсь от написания кода для поддержки тех из них, которые в данный момент еще не представлены на практике. Тем не менее, проектирование от контекста с применением шаблонов позволяет предвидеть и учесть появление возможных изменений, создавая систему, продуманно разделенную на классы и заранее приспособленную к ожидаемым изменениям. Шаблоны проектирования помогают понять, в каких местах возможны изменения, но не дают конкретных указаний о том, какими они будут. Однако хорошо продуманный интерфейс будет не только эффективно работать с уже существующими вариациями, но позволит учесть потребности новых потенциальных требований.

Шаблон *Abstract Factory* — еще один хороший пример проектирования от контекста. Вполне очевидно, что объект-фабрика некоторого типа будет использоваться для координации процесса создания семейств (или наборов) экземпляров объектов. Однако существует несколько различных способов реализовать эту задачу (табл. 13.1).

Таблица 13.1. Возможные варианты реализации шаблона *Abstract Factory*

| Вариант | Описание |
|--|--|
| Использование производных классов | Классическая реализация шаблона <i>Abstract Factory</i> требует определения производных классов для каждого из существующих наборов объектов. Это несколько громоздкий вариант, но он имеет весомые преимущества, позволяя добавлять новые классы, не затрагивая ни один из уже существующих |
| Использование одного объекта с переключателями | Если согласиться на изменение класса <i>Abstract Factory</i> по мере необходимости, то можно просто создать один объект, который будет включать все правила. Хотя этот подход противоречит принципу открытости-закрытости, все правила будут реализованы в одном месте, и такую систему будет достаточно легко обслуживать |
| Использование файла конфигурации и переключателей | Это более гибкий способ по сравнению с предыдущим, но здесь так же время от времени потребуется вносить изменения в программный код |
| Использование файла конфигурации совместно с механизмом RTTI | Механизм RTTI (RunTime-Type-Identification — определение типа во время выполнения) включает средства создания экземпляров объектов с выбором их типа на основании имени объекта, помещенного в строковую переменную. Реализации этого типа присуща максимальная гибкость, так как новые классы и новые комбинации могут быть добавлены в систему без изменения какого-либо программного кода |

Какие же рекомендации можно дать по выбору оптимального варианта реализации шаблона Abstract Factory? Ответ прост – решение нужно принимать исходя из того контекста, в котором он присутствует. Каждый из четырех вариантов реализации имеет свои преимущества в зависимости от следующих факторов.

- Вероятность будущих изменений.
- Важность сохранения неизменности существующей системы.
- Доступность для модификации классов, входящих в отдельные семейства (кто является их создателем – вы или другая группа программистов).
- Используемый язык программирования.
- Наличие и доступность базы данных или файла конфигурации.

Безусловно, этот список не полон, так же, как и список вариантов реализации. Однако каждому должно быть понятно, что попытка решить, как следует реализовать шаблон Abstract Factory, без учета того, как создаваемая система будет использоваться (т.е. без понимания контекста), выглядит, по меньшей мере, глупо.

Как принимаются проектные решения

Делая выбор между альтернативными вариантами реализации, многие разработчики стремятся получить ответ на вопрос: "Какая из возможных реализаций является лучшей?". Однако на самом деле так ставить вопрос нельзя. Проблема заключается в том, что очень редко одна реализация бывает лучше другой во всех отношениях. Предпочтительнее рассматривать каждую альтернативу в отдельности, задаваясь следующим вопросом: "При каких обстоятельствах данная альтернатива будет лучше, чем другие?". Затем следует ответить на такой вопрос: "Какие из этих обстоятельств более всего характерны для заданной проблемной области?". При необходимости легко можно остановиться и вернуться на шаг назад. Подобный подход заостряет внимание проектировщика на возможных вариациях и проблемах масштабирования системы в заданной проблемной области.

Шаблон Adapter иллюстрирует принцип проектирования от контекста потому, что сам он почти всегда обнаруживается в пределах контекста. По определению шаблон Adapter применяется для приведения существующего интерфейса к некоторому другому интерфейсу. Напрашивается следующий вопрос: "Как узнать, к какому виду следует привести существующий интерфейс?". Как правило, до окончательного формирования контекста (определения, к какому именно классу требуется адаптация) ответить на этот трудно.

Выше уже обсуждалось, как шаблон Adapter может быть использован для адаптации класса к роли, предусмотренной для него шаблоном. В частности, при решении проблемы поддержки в приложении нескольких версий САПР потребовалось адаптировать определенную существующую реализацию к виду, необходимому для использования ее в шаблоне Bridge.

Шаблон Facade в терминах контекста очень напоминает шаблон Adapter. Чаще всего он присутствует в контексте других шаблонов или классов. Следовательно, прежде чем приступить к разработке интерфейса, необходимо подождать, пока не будет установлено, кто же его будет использовать.

При первых попытках использования шаблонов проектирования я полагал, что всегда можно отыскать шаблон, который создает контекст для других шаблонов. Во всяком случае, Александеру в его книге всегда удавалось сделать это, правда он имел дело с шаблонами в области архитектуры. Поскольку множество людей принимало участие в обсуждении проблемы создания языка шаблонов для разработки программного обеспечения, я также задумался: "Почему бы и мне не попробовать?". Ведь кажется абсолютно очевидным, что шаблоны Adapter и Facade всегда будут определяться в контексте чего-то другого.

Однако это оказалось не так. Те разработчики программного обеспечения, которые, как и я, занимаются преподаванием, обладают одним большим преимуществом. Оно состоит в том, что преподаватели принимают участие в гораздо большем количестве проектов, чем обычные разработчики. В начале моей преподавательской деятельности, связанной с шаблонами проектирования, я полагал, что в последовательности определения контекста шаблоны Adapter и Facade всегда рассматриваются после других шаблонов, не связанных с созданием объектов. Чаще всего именно так и происходит. Однако некоторые системы включают требование создания некоторого специфического интерфейса. В этом случае шаблон Facade или Adapter (единственный из многих шаблонов, присутствующих в системе) может оказаться в системе шаблоном высшего уровня.

Принцип включения вариаций

Несколько человек независимо указали на определенное сходство всех моих проектов, с которыми им приходилось сталкиваться. Дело в том, что иерархия наследования классов в моих проектах редко содержит более двух уровней в глубину. Так происходит потому, что мои проекты имеют типичную для шаблонов проектирования структуру из двух уровней для основных и абстрактных классов. (Правда, существуют и исключения – например, шаблон Decorator, обсуждаемый в главе 15, использует три уровня классов.)

Основная причина заключается в том, что при проектировании я придерживаюсь правила никогда не создавать классов, включающих несколько подверженных вариациям элементов, которые так или иначе связаны между собой. Обсуждавшиеся нами выше шаблоны демонстрируют различные способы эффективного включения вариаций.

Шаблон Bridge является собой превосходный пример включения вариаций. Все реализации, присутствующие в шаблоне Bridge, различны, но доступ к ним организуется через общий интерфейс. Новые реализации легко могут быть осуществлены в пределах этого интерфейса.

Шаблон Abstract Factory включает вариации в отношении наборов или семейств тех объектов, экземпляры которых создаются. Существует несколько различных путей реализации этого шаблона. Хочу особо обратить ваше внимание на то, что даже если первоначально был выбран некоторый вариант реализации, а позднее было установлено, что имеется другой, лучший путь, реализация шаблона может быть изменена без оказания какого-либо влияния на остальные части системы (интерфейс фабрики остается неизменным, меняется только способ ее реализации). Таким образом, сам принцип построения шаблона Abstract Factory скрывает все вариации в отношении того, как объекты создаются.

Шаблон Adapter – это инструмент, обеспечивающий использование различных по происхождению объектов через общий интерфейс. Это часто бывает необходимо в отношении интерфейсов, вызываемых из многих шаблонов. Таким образом, шаблон Adapter предназначен для скрытия вариаций в интерфейсах классов.

Шаблон Facade, как правило, не включает изменений. Однако практика дает множество примеров использования шаблона Facade для работы с конкретными подсистемами. В этом случае при появлении новой подсистемы для нее строится собственный шаблон Facade с тем же самым интерфейсом. Этот новый класс представляет собой комбинацию шаблонов Facade и Adapter. Однако, если изначально эти шаблоны использовались для упрощения, то теперь они позволяют сохранить прежний интерфейс и избежать изменения существующих клиентских объектов. Подобное применение шаблона Facade позволяет скрыть вариации в используемых подсистемах.

Однако шаблоны предназначены не только для включения вариаций. Они также определяют отношения между отдельными вариациями. Подробнее об этом речь пойдет в следующих главах. В отношении шаблона Bridge можно дополнительно указать, что он не только определяет и включает вариации в абстракции и реализации, но и определяет отношения между ними.

Резюме

В этой главе мы обсудили, как в шаблонах проектирования иллюстрируется применение двух мощных стратегий проектирования:

- проектирование от контекста;
- включение вариаций в классы.

Использование этих стратегий позволяет отложить принятие решения до тех пор, пока не будут выявлены все возможные варианты. Тщательный анализ контекста решаемой задачи позволяет найти лучшие проектные решения.

Посредством включения вариаций в классы удается учесть потенциальные изменения, которые могут остаться необнаруженными, если предварительно не взглянуть на проект с более общей точки зрения. Это особенно важно для таких проектов, которые не обеспечиваются всеми требуемыми ресурсами в полном объеме (другими словами, для всех проектов). Корректное включение вариаций позволяет ограничиться реализацией только тех функций, которые требуются на текущий момент, не ставя под угрозу сохранение качества проекта в будущем. Не стоит забывать, что попытки выявить *все* потенциальные вариации и обеспечить их поддержку в системе обычно ведут не к созданию лучшей системы, а к краху проекта вообще. Это явление обычно называют параличом от анализа.

Глава 14

Шаблон Strategy

Введение

В этой главе мы познакомимся с новым практическим примером, взятым из области электронной коммерции. В уже существующую систему поддержки розничных продаж через Internet будет предложено внести определенное изменение. Решая поставленную задачу, мы познакомимся с шаблоном *Strategy* (стратегия). В каждой последующей главе, вплоть до главы 20, *Матрица анализа*, наш новый пример будет последовательно усложняться.

Здесь мы выполним следующее.

- Обсудим оптимальный подход к обработке новых требований к существующей системе.
- Познакомимся с новым учебным примером.
- Опишем шаблон *Strategy* и рассмотрим, как его можно использовать для обработки новых требований в нашем примере.
- Проанализируем ключевые особенности шаблона *Strategy*.

Оптимальный подход к обработке новых требований

Как в обычной жизни, так и при проектировании программных приложений каждому из нас неоднократно приходилось искать оптимальный подход к решению задачи или устранению проблемы. Большинство из нас на собственном опыте знает, что решения, обеспечивающие моментальную выгоду, часто приводят к серьезным затруднениям в будущем. Например, каждый водитель знает, что после определенного пробега масло в автомобиле обязательно следует поменять. Конечно, нет необходимости менять масло через каждые 5 000 км, однако откладывать полную смену масла в двигателе до достижения пробега в 50 000 км нельзя (иначе необходимость замены масла отпадет сама собой – автомобиль просто выйдет из строя!). Другой пример – представьте себе письменный стол. Очень удобно хранить все нужные бумаги и канцелярские принадлежности под рукой, прямо на его поверхности. Однако это удобство будет сохраняться недолго, и через некоторое время стол окажется завален грудами бумаг, папок, книг и тому подобного, в которых найти что-либо будет просто невозможно. Кстати, различные катастрофы чаще всего являются отдаленным следствием псевдо-оптимальных решений, принятых на скорую руку, без учета длительной перспективы.

К большому сожалению, многие специалисты в области разработки программного обеспечения все еще не усвоили этот урок. Множество проектов разрабатывалось с ориентацией на решение только конкретных, сиюминутных задач без учета проблемы сопровождения системы в будущем. Существует несколько предпосылок, способствующих сохранению тенденции к игнорированию таких важнейших долгосрочных аспектов создаваемых проектов, как простота его сопровождения или удобство внесения изменений. Чаще всего указывают на следующие причины.

- Очень сложно предвидеть, как требования к системе будут изменяться в будущем.
- Если заняться выяснением, как требования к системе будут изменяться в будущем, этап анализа не закончится никогда.
- Если начать писать создаваемое программное обеспечение так, чтобы оно позволяло легко добавлять в него новую функциональность, проектирование не закончится никогда.
- У нас на это нет ни времени, ни денег.

Существуют две крайности.

- Чрезмерное углубление в анализ и разнообразные аспекты проектирования, что обычно называют "параличом от анализа".
- Принятие скороспелого решения с немедленным переходом к кодированию без какого-либо анализа долгосрочных аспектов проекта. В результате, очень скоро потребуется начать новый проект, так как подобная недальновидность порождает слишком много проблем.

Поскольку руководство обычно больше интересуется сроками сдачи системы, а не вопросами ее сопровождения, то такой результат не удивителен. После недолгого размышления становится совершенно очевидным, что именно удерживает разработчиков программного обеспечения от проведения анализа возможных альтернатив. Не вызывает сомнения, что большинство из них просто убеждены в том, что проектирование с поддержкой возможных будущих изменений непременно потребует дополнительных затрат.

Но это утверждение вовсе необязательно будет справедливо. В действительности, истинно как раз обратное утверждение. Если взглянуть на систему в целом и попытаться определить, как она будет изменяться в будущем, обычно удается найти лучшее проектное решение, реализация которого потребует фактически такого же количества времени, как и при обычной практике проектирования.

Именно такой подход к проектированию — с предварительным анализом и учетом возможных изменений — мы используем при обсуждении приведенного ниже учебного примера. Здесь очень важно еще раз обратить ваше внимание на то, что мы будем заранее ожидать появления изменений, и задача наша состоит в том, чтобы выяснить, где они будут происходить, а не в том, чтобы определить, какими именно они будут. Данный подход основан на принципах, предложенных в книге "банды четырех".

- "Проектируйте интерфейс, а не его реализацию."¹

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 18.

- "Отдавайте предпочтение композиции объектов, а не наследованию классов."²
- "Выясните, что может быть подвержено изменениям в вашем проекте." Этот подход – прямая противоположность фокусированию внимания на причинах, способных вызвать перепроектирование системы. Вместо того, чтобы искать то, что способно привести к изменению проекта, следует отобрать то, для чего нужно обеспечить возможность изменения в будущем без необходимости перепроектирования. Суть подхода заключается в инкапсуляции изменяемого на концептуальном уровне, что является лейтмотивом многих шаблонов проектирования.³

Я настоятельно вам рекомендую, столкнувшись с необходимостью изменения программного кода в связи с появлением новых требований, проанализировать возможность применения приведенных ниже стратегий. Использование этих стратегий не приведет к существенному удорожанию проекта или его реализации, но позволит получить значительные преимущества в будущем.

Однако я не предлагаю следовать этим стратегиями вслепую. Всегда можно оценить качество альтернативного варианта, проверив, насколько полно он отвечает показателям хорошего объектно-ориентированного проекта. Такой же, в сущности, подход был использован при выведении шаблона Bridge в главе 9, *Шаблон Bridge*, где мы сравнивали качество альтернативных вариантов на основании того, какой из них лучше соответствует принципам объектно-ориентированного программирования.

Исходные требования к учебному проекту

Предположим, что перед нами поставлена задача разработать систему поддержки электронной розничной торговли в пределах Соединенных Штатов Америки. Общая структура приложения включает объект-контроллер, предназначенный для обработки поступающих запросов на продажу. Он определяет момент поступления запроса на формирование заказа и передает поступивший запрос объекту выписки счета для дальнейшей обработки.

Общий вид исходного варианта системы представлен на рис. 14.1.

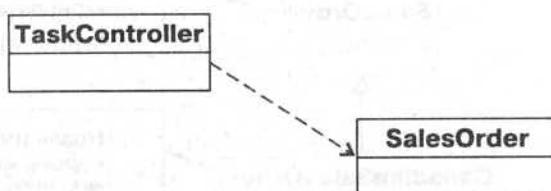


РИС. 14.1. Схема обработки заказа в системе розничной электронной торговли

² Там же, с. 20.

³ Там же, с. 29.

Класс **SalesOrder** имеет следующие функции.

- Предоставление графического интерфейса для заполнения заказа.
- Вычисление суммы налога.
- Обработка заказа и печать накладной на продажу.

Некоторые из этих функций, вероятно, следует реализовать с помощью других объектов. Например, класс **SalesOrder** не обязательно должен непосредственно заниматься выводом на печать, скорее его назначение состоит в хранении информации о поступившем заказе на продажу. Отдельные объекты класса **SalesOrder** для распечатки накладной на продажу могут обращаться к одному и тому же объекту класса **Salesticket**.

Обработка новых требований

Предположим, что в процессе работы над этим приложением было выдвинуто новое требование об изменении способа начисления налога. Теперь система должна будет поддерживать начисление налога и для заказов тех клиентов, которые находятся за пределами Соединенных Штатов. Как минимум, потребуется поддержка новых правил начисления таких налогов.

Как организовать в системе поддержку этих новых правил? Можно попробовать еще раз воспользоваться уже существующим классом **SalesOrder** и обрабатывать данную ситуацию просто как новый вид коммерческого заказа с отличающимся набором правил налогообложения. Например, для обработки заказов из Канады можно создать новый класс с именем **CanadianSalesOrder**, производный от класса **SalesOrder**, в котором переопределяются правила налогообложения. Данный вариант решения представлен на рис. 14.2.

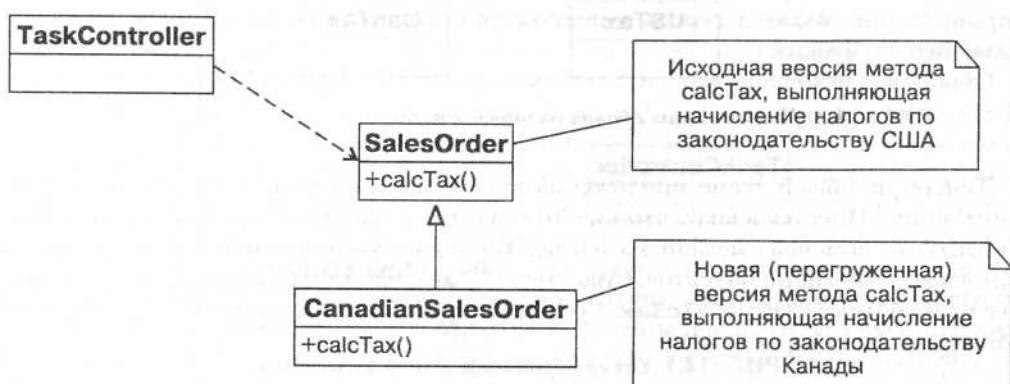


РИС. 14.2. Возможный вариант схемы обработки заказа в системе розничной электронной торговли

ции объектов, а не наследованию классов".⁴ В решении на рис. 14.2 использован прямо противоположный подход! Другими словами, изменение в налоговых правилах обрабатывается здесь с использованием механизма наследования посредством определения производного класса, включающего поддержку новых правил.

Какое другое решение можно предложить? Следуя изложенному выше правилу, нужно попытаться найти то, что является в проекте изменяемым, и инкапсулировать эту концепцию.⁵

Требуемый результат достигается в два этапа.

1. Найти то, что изменяется, и инкапсулировать это в соответствующий специализированный класс.
2. Поместить этот класс в другой класс.

В нашем примере уже известно, что изменяемой концепцией являются правила налогообложения. Инкапсуляция в этом случае подразумевает создание абстрактного класса, определяющего решение поставленной задачи на концептуальном уровне, с последующим созданием конкретных производных классов для каждого из существующих вариантов. Другими словами, необходимо создать абстрактный класс **CalcTax**, определяющий интерфейс, требуемый для решения поставленной задачи, а затем создать конкретные классы, производные от него, для каждой существующей версии. Схематично данное решение представлено на рис. 14.3.

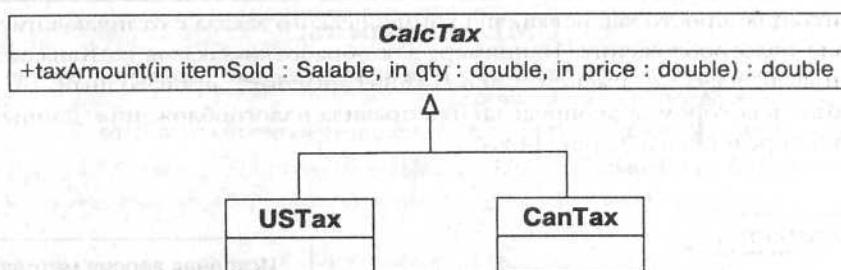


РИС. 14.3. Инкапсуляция правил налогообложения

Теперь на общей схеме приложения вместо наследования классов используется композиция. Имеется в виду, что вместо создания различных версий класса обработки заказа (с помощью механизма наследования) включение вариации выполнено с помощью композиции объектов. Существует только один класс **SalesOrder**, который содержит объект класса **CalcTax**, предназначенный для скрытия и обработки возможных вариаций (рис. 14.4).

⁴ Там же, с. 20.

⁵ Там же, с. 29.

В основу шаблона Strategy положено несколько принципов.

- Объекты обладают обязательствами.
- Различные специфические реализации этих обязательств проявляются за счет использования полиморфизма.
- Существует потребность в управлении несколькими различными реализациями того, что концептуально является одним и тем же алгоритмом.
- Следует считать хорошим стилем проектирования отделение существующих в проблемной области поведений друг от друга – т.е. уменьшение связности между ними. Это позволяет вносить изменения в класс, ответственный за некоторое поведение, без какого-либо неблагоприятного воздействия на другие классы.

Основные характеристики шаблона Strategy

| | |
|----------------|--|
| Назначение | Позволяет использовать различные бизнес-правила или алгоритмы в зависимости от контекста |
| Задача | Выбор алгоритма, который следует применить, в зависимости от типа выдавшего запрос клиента или обрабатываемых данных. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону Strategy |
| Способ решения | Отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста |
| Участники | <ul style="list-style-type: none"> • Класс Strategy определяет, как будут использоваться различные алгоритмы • Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы • Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посыпать запросы классу Context). Класс Context пересыпает классу Strategy запрос, поступивший от его класса-клиента |
| Следствия | <ul style="list-style-type: none"> • Шаблон Strategy определяет семейство алгоритмов • Это позволяет отказаться от использования переключателей и/или условных выражений • Вызов всех алгоритмов должен осуществляться стандартным образом (все они должны иметь один и тот же интерфейс). Взаимодействие между классами ConcreteStrategy и Context может потребовать введения в класс Context дополнительных методов типа <code>getState</code> |
| Реализация | <p>Класс, который использует алгоритм (Context), включает абстрактный класс (Strategy), обладающий абстрактным методом, определяющим способ вызова алгоритма. Каждый производный класс реализует один требуемый вариант алгоритма.</p> <p><i>Замечание.</i> Метод вызова алгоритма не может быть абстрактным, если требуется реализовать некоторое поведение, принимаемое по умолчанию.</p> |

Замечание. В исходном варианте шаблона *Strategy* ответственность за выбор конкретной реализации возлагается на объект-клиент — т.е. на контекст шаблона *Strategy*

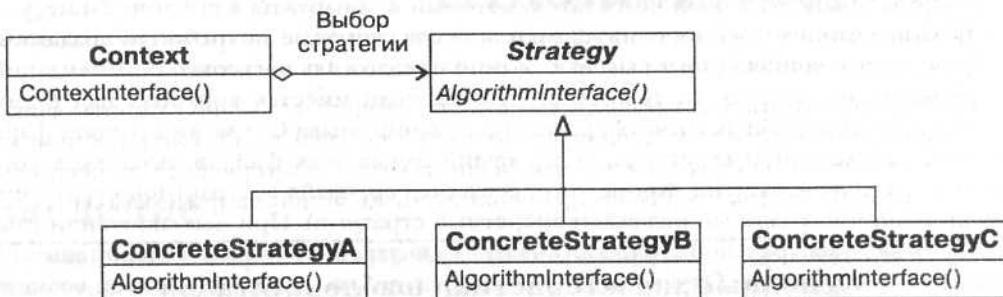


РИС. 14.6. Стандартное упрощенное представление шаблона *Strategy*

Дополнительные замечания о шаблоне *Strategy*

Однажды, когда на занятиях по изучению шаблонов проектирования обсуждался пример с системой электронной торговли, кто-то в аудитории задал мне вопрос: "А знаете ли вы, что в Англии люди по достижении определенного возраста не платят налог при покупке продовольственных товаров?". Эта информация была для меня новостью, и вполне естественно, что интерфейс объекта `CalcTax` не позволял обрабатывать подобные ситуации. Существует по меньшей мере три способа решения данной проблемы.

1. Передавать сведения о возрасте покупателя от объекта класса `Customer` объекту класса `CalcTax` и использовать их по мере необходимости.
2. Выбрать более общий подход: передавать объекту класса `CalcTax` весь объект класса `Customer` в целом и опрашивать его в случае необходимости.
3. Еще повысить уровень обобщения, передавая объекту класса `CalcTax` ссылку на объект класса `SalesOrder` (т.е. его указатель `this`), и позволить объекту класса `CalcTax` опрашивать его.

Хотя для обработки подобной ситуации обязательно потребуется изменить классы `SalesOrder` и `CalcTax`, совершенно очевидно, как это можно сделать. Маловероятно, что выполнение требуемых действий вызовет появление в системе каких-либо проблем.

Формально шаблон *Strategy* представляет собой средство инкапсуляции алгоритмов. Однако на практике он может использоваться для инкапсуляции правил фактически любого вида. В общем случае, если на этапе анализа требований становится известно о существовании различных бизнес-правил, применяемых в различных ситуациях, обязательно следует рассмотреть возможность применения шаблона *Strategy*, эффективно обрабатывающего вариации такого рода.

Шаблон Strategy требует, чтобы алгоритмы (бизнес-правила) были инкапсулированы вне класса, который их использует (**Context**). Это означает, что информация, необходимая для выполнения этих алгоритмов, должна быть передана или получена каким-то иным способом.

Единственный серьезный недостаток, который я обнаружил в шаблоне Strategy, – это большое количество дополнительных классов, которые потребуется создавать. В случае, когда овчинка стоит выделки, можно предложить несколько рекомендаций по уменьшению количества требуемой работы, если имеется контроль над всеми стратегиями. В подобных случаях при использовании языка C++ можно создать файл заголовка абстрактной стратегии, содержащий имена всех файлов заголовков конкретных классов стратегий. Кроме того, создается cpp-файл абстрактной стратегии, содержащий программный код всех конкретных стратегий. При использовании языка Java в классе абстрактной стратегии создаются внутренние классы, содержащие код конкретных стратегий. Однако этого не следует делать, если отсутствует возможность контроля над всеми стратегиями – т.е. если некоторые алгоритмы будут реализовать другие программисты.

Резюме

Шаблон Strategy – это способ определить семейство алгоритмов. Концептуально все эти алгоритмы решают одну и ту же задачу, но различаются между собой способом ее решения.

Мы рассмотрели пример, в котором используется семейство алгоритмов начисления налогов. В системе поддержки международной электронной розничной торговли необходимо использовать различные алгоритмы начисления налогов для покупателей из различных стран. Шаблон Strategy позволяет инкапсулировать эти правила в одном абстрактном классе, от которого производится требуемое семейство конкретных классов.

За счет определения различных вариантов выполнения алгоритма как производных от одного абстрактного класса главный модуль (в нашем примере это класс **SalesOrder**) сможет не принимать во внимание, какую именно версию алгоритма он фактически использует. Такой подход упрощает подключение новых вариантов алгоритма, но, одновременно, требует некоторых мер по управлению ими. Об этом мы поговорим позже, в главе 20, *Матрица анализа*.

Глава 17

Шаблон Observer

Введение

В этой главе мы продолжим обсуждение нашего учебного примера – приложения поддержки электронной розничной торговли, начатое в главах 14–16.

Здесь мы выполним следующее.

- Рассмотрим схему классификации шаблонов.
- Познакомимся с шаблоном Observer в процессе обсуждения дополнительных требований, предъявленных к нашему учебному примеру.
- Применим данный шаблон в системе поддержки электронной торговли.
- Обсудим характеристики этого шаблона.
- Рассмотрим ключевые особенности шаблона Observer.
- Познакомимся с моим опытом применения шаблона Observer на практике.

Категории шаблонов

Количество уже существующих шаблонов достаточно велико. Поэтому для их упорядочения "банда четырех" предложила разделить все шаблоны на три категории, как показано в табл. 17.1.¹

Таблица 17.1. Категории шаблонов

| Категория | Назначение | Примеры в этой книге | Для чего используются |
|---------------|---|--|---|
| Структурные | Связывают между собой существующие объекты | <ul style="list-style-type: none">• Facade (глава 6)• Adapter (глава 7)• Bridge (глава 9)• Decorator (глава 15) | Приведение интерфейсов Связывание реализации с абстракцией |
| Поведенческие | Определяют способы проявления гибкого (изменяющегося) поведения | <ul style="list-style-type: none">• Strategy (глава 14) | Скрытие вариаций |

¹ Gamma, E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 10

Окончание таблицы

| Категория | Назначение | Примеры в этой книге | Для чего используются |
|-----------|--|---|-------------------------------|
| Создающие | Управляют созданием экземпляров объектов | <ul style="list-style-type: none"> • Abstract Factory (глава 10) • Singleton (глава 16) • Double-Checked Locking (глава 16) • Factory Method (глава 19) | Создание экземпляров объектов |

Впервые приступив к изучению шаблонов проектирования, я очень удивился тому, что шаблоны Bridge и Decorator были отнесены к категории структурных, а не поведенческих шаблонов. На первый взгляд, не вызывало сомнения, что они используются для реализации различных типов поведения. Но, как выяснилось впоследствии, я просто неправильно понял систему классификации "банды четырех". Структурные шаблоны предназначены для связывания между собой уже существующих функций. Работая с шаблоном Bridge, мы, как правило, начинаем с функций абстракции и реализации, а затем связываем их между собой. Шаблон Decorator имеет дело с уже существующим функциональным классом и предназначен для "декорирования" его дополнительными функциями.

Я пришел к заключению, что имеет смысл выделить еще одну, четвертую, категорию шаблонов. К ней я отношу шаблоны, основное назначение которых – уменьшить связанность объектов друг с другом, и, следовательно, повысить возможности масштабируемости системы и ее гибкость. Я назвал эту категорию *развязывающими шаблонами*. Поскольку по классификации "банды четырех" большинство подобных шаблонов принадлежат к категории поведенческих, их можно было бы считать подмножеством этой категории. Я решил выделить их в четвертую категорию только потому, что намеревался в этой книге отразить мой собственный взгляд на шаблоны проектирования, сосредоточив внимание на мотивах их определения – в данном случае это развязывание объектов в системе.

Я не буду более задерживаться на обсуждении различных аспектов классификации шаблонов, поскольку это требует достаточно глубокого понимания принципов их построения и работы.

В этой главе мы познакомимся с шаблоном Observer, который можно считать лучшим примером развязывающего шаблона среди всех прочих. По классификации "банды четырех" шаблон Observer относится к группе поведенческих.

Предъявление новых требований к системе электронной торговли

Предположим, что при разработке нашего приложения поддержки электронной торговли было выдвинуто очередное новое требование. Выяснилось, что необходимо выполнять перечисленные ниже действия всякий раз, когда к системе подключается новый клиент.

Шаблон Observer

Согласно определению "банда четырех" назначение шаблона Observer (наблюдатель) состоит в следующем.

Определение зависимости типа "один ко многим" между объектами, выполненное таким образом, что при изменении состояния одного объекта все зависимые объекты были бы уведомлены об этом и обновлены автоматически.²

Весьма распространена ситуация, при которой существует набор объектов, которые должны получать уведомление всякий раз, когда происходит некоторое событие. Необходимо, чтобы это уведомление выполнялось автоматически. Кроме того, желательно обойтись без внесения изменений в оповещающий объект при каждом изменении в наборе объектов, получающих уведомления. (В противном случае ситуация походила бы на требование внесения изменений в радиопередатчик всякий раз, когда в городе появляется новый автомобиль с радиоприемником.) Таким образом, мы должны снизить уровень связанности между оповещающим и извещаемыми объектами.

Шаблон Observer относится к наиболее широко используемым шаблонам. Иначе его иногда называют *Dependents* (зависимости) или *Publish-Subscribe*³ (публикация-подписка). Он является аналогом процесса извещения в технологии COM. В языке Java данный шаблон реализован с помощью интерфейса **Observer** и класса **Observable** (подробнее об этом речь пойдет ниже). В экспертных системах, использующих базы правил, шаблон Observer часто реализуется в демонах правил.

Применение шаблона Observer в системе электронной торговли

Наш подход к решению проблемы состоит в том, чтобы найти то, что может быть изменяемым в системе, а затем попытаться инкапсулировать эти вариации. В нашем учебном примере изменяемым может быть следующее.

- **Различные виды объектов.** Существует список объектов, которые необходимо уведомить об изменении состояния системы. Эти объекты, как правило, принадлежат к различным классам.
- **Различные интерфейсы.** Поскольку извещаемые объекты принадлежат к различным классам, они, вероятнее всего, будут обладать различными интерфейсами.

Прежде всего, выявим все объекты, которые должны получать уведомления. Назовем их *наблюдателями* (*observer*), так как они находятся в состоянии ожидания некоторого события, которое должно произойти.

Желательно, чтобы все эти объекты имели одинаковый интерфейс. Если интерфейсы не будут одинаковыми, придется модифицировать наблюдаемый *субъект* – т.е. тот объект, который инициирует ожидаемое событие (например, объект класса *Customer*), чтобы он мог обращаться к объектам-наблюдателям различного типа.

² Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading MA: Addison-Wesley, 1995, с. 293.

³ Там же, с. 293.

Если все объекты-наблюдатели будут одного типа, то субъект легко сможет уведомить каждый из них. Чтобы достичь этой цели можно поступить следующим образом.

- В языке Java, вероятно, лучше всего использовать интерфейс (чтобы повысить гибкость или просто по необходимости).
- В языке C++ можно использовать одиночное или множественное наследование — исходя из конкретных обстоятельств.

Обычно требуется, чтобы объекты-наблюдатели знали, за кем они должны наблюдать, а субъект был бы освобожден от необходимости знать, какие именно объекты-наблюдатели от него зависят. Чтобы достичь этого, необходимо предоставить каждому из объектов-наблюдателей возможность зарегистрироваться у субъекта. Поскольку все объекты-наблюдатели имеют один и тот же тип, в класс-субъект необходимо добавить два метода.

- Метод `attach(Observer)`. Добавляет заданный объект класса `Observer` к списку объектов-наблюдателей данного субъекта.
- Метод `detach(Observer)`. Удаляет заданный объект класса `Observer` из списка объектов-наблюдателей данного субъекта.

Теперь, когда класс `Subject` может регистрировать объекты класса `Observer`, очень просто известить всех зарегистрировавшихся объектов-наблюдателей о наступлении ожидаемого события. Для этого в каждом конкретном типе класса `Observer` реализуется метод `update` (обновить). В классе `Subject` реализуется метод `notify`, который просматривает список зарегистрировавшихся объектов-наблюдателей и вызывает метод `update` каждого из них. Метод `update` объектов-наблюдателей должен включать программный код обработки наступления ожидаемого события.

Однако просто уведомить каждый объект-наблюдатель о наступлении события будет недостаточно. Объекту-наблюдателю может понадобиться дополнительная информации о событии, а не просто извещение о том, что оно произошло. Поэтому в класс-субъект следует добавить еще один или несколько методов, позволяющих объектам-наблюдателям получить ту информацию, в которой они нуждаются. Это решение представлено на рис. 17.2.

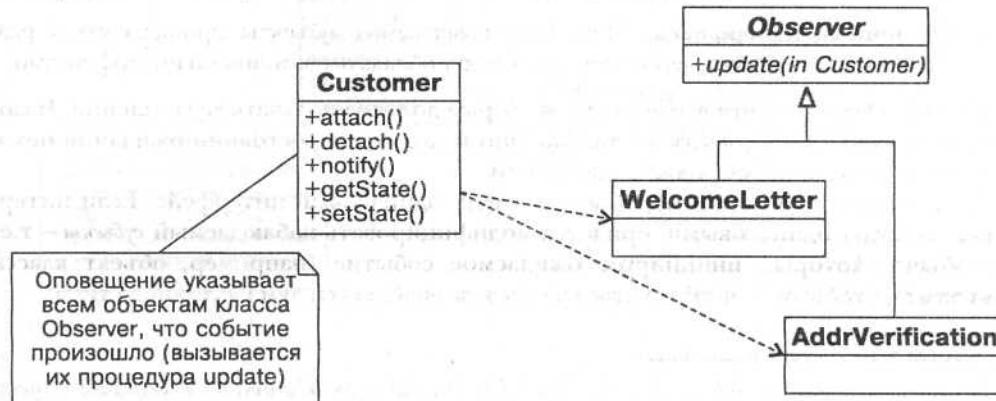


РИС. 17.2. Реализация взаимодействия классов `Customer` и `Observer`

На рис. 17.2 классы взаимодействуют между собой следующим образом.

1. Объекты класса **Observer** подключаются к классу **Customer** при их инициализации. Если объекту класса **Observer** необходима дополнительная информация от субъекта (объект класса **Customer**), он должен передать вызываемому объекту ссылку на свой метод `update`.
2. Каждый раз при добавлении нового объекта класса **Customer**, его метод `notify` вызывает все зарегистрированные объекты класса **Observer**.

Каждый объект класса **Observer** вызывает метод `getState` (класс **Customer**) для получения информации о вновь добавленном клиенте – это необходимо ему, чтобы выяснить, какие действия следует предпринять. *Замечание.* Обычно для получения необходимой информации требуется несколько методов.

Обратите внимание на то, что в нашем примере для добавления и удаления оповещаемых объектов в списке используются *статические* методы. Причина в том, что объекты-наблюдатели должны быть уведомлены обо *всех* новых клиентах (т.е. создаваемых объектах класса **Customer**). Вместе с уведомлением наблюдателю передается ссылка на вновь созданный объект описания клиента.

Код реализации описанного выше подхода представлен в листинге 17.1 (частично).

Листинг 17.1. Реализация шаблона Observer на языке Java

```

class Customer {
    static private Vector myObs;
    static {
        myObs= new Vector();
    }
    static void attach(Observer o){
        myObs.addElement(o);
    }
    static void detach(Observer o){
        myObs.remove(o);
    }
    public String getState () {
        // Здесь могут вызываться другие методы,
        // представляющие требуемую информацию
    }

    public void notifyObs () {
        for (Enumeration e =
            myObs.elements();
            e.hasMoreElements() ;) {
            ((Observer) e).update(this);
        }
    }
}

abstract class Observer {
    public Observer () {
        Customer.attach( this);
    }
    abstract public void
        update(Customer myCust);
}

class POVerification extends Observer {
    public AddrVerification () {

```

```

    super();
}
public void update (
Customer myCust) {
    // Здесь выполняется проверка адресов.
    // Дополнительная информация о клиенте может быть
    // получена с помощью метода myCust
}
}

class WelcomeLetter extends Observer {
public WelcomeLetter () {
    super();
}
public void update (Customer myCust) {
    // Здесь выполняется отправка приветственного письма.
    // Дополнительная информация о клиенте может быть
    // получена с помощью метода myCust
}
}
}

```

Данный подход позволяет добавлять новые объекты-наблюдатели без какого-либо влияния на любые уже существующие классы. Он также позволяет поддерживать в системе низкий уровень связанности. Организованная подобным образом система работает так, как если бы все объекты в ней отвечали только сами за себя.

Что произойдет в случае предъявления к системе нового требования? Предположим, возникла необходимость посылать письмо с купонами тем клиентам, которые находятся в пределах 20 миль от одного из складов компании.

Чтобы реализовать это требование, достаточно просто определить новый класс-наблюдатель, выполняющий отправку купонов. Он должен отправлять купоны только для тех новых клиентов, которые расположены не далее указанного расстояния от ближайшего из складов компании. Назовем этот класс **BrickAndMortar** и определим его как один из объектов-наблюдателей класса **Customer**. Данное решение представлено на рис. 17.3.

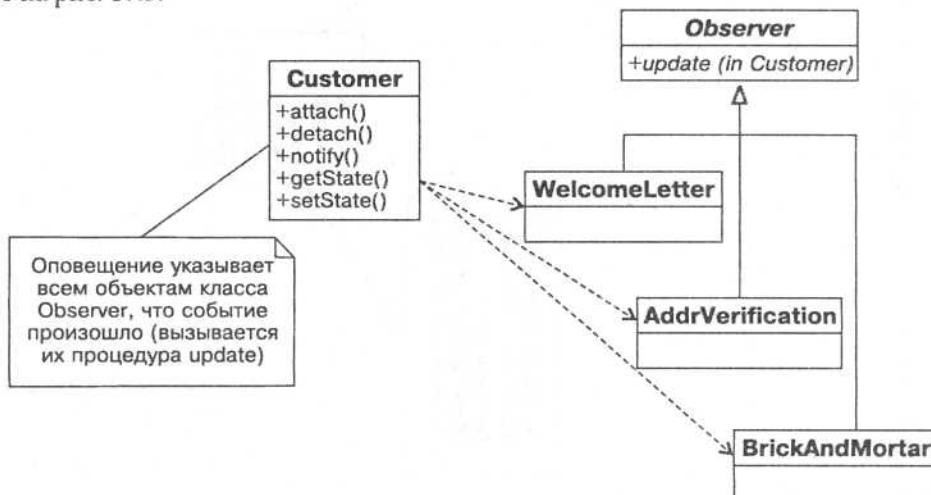


РИС. 17.3. Добавление нового объекта-наблюдателя *BrickAndMortar*

Иногда класс, который должен стать новым наблюдателем, может уже существовать в системе. В этом случае, вероятно, будет предпочтительнее не вносить в него никаких изменений. Существует простой путь решить эту проблему: достаточно адаптировать существующий класс с помощью уже знакомого нам шаблона Adapter. Пример подобного решения приведен на рис. 17.4.

Класс **Observable** — замечание для разработчиков, использующих язык Java

Шаблон Observer настолько полезен, что один из пакетов языка Java включает его встроенную реализацию. В данном случае шаблон составлен из класса **Observable** и интерфейса **Observer**. Класс **Observable** выполняет роль субъекта из описания схемы шаблона, представленного в книге "банды четырех". В качестве методов `attach`, `detach` и `notify` в языке Java используются, соответственно, методы `addObserver`, `deleteObserver` и `notifyObservers` (в Java используется и метод `update`). Язык Java также предоставляет несколько других методов, упрощающих работу с данными классами. (Дополнительные сведения об API языка Java для классов **Observer** и **Observable** можно найти в Internet по адресу <http://java.sun.com/j2se/1.3/docs/api/index.html>.)

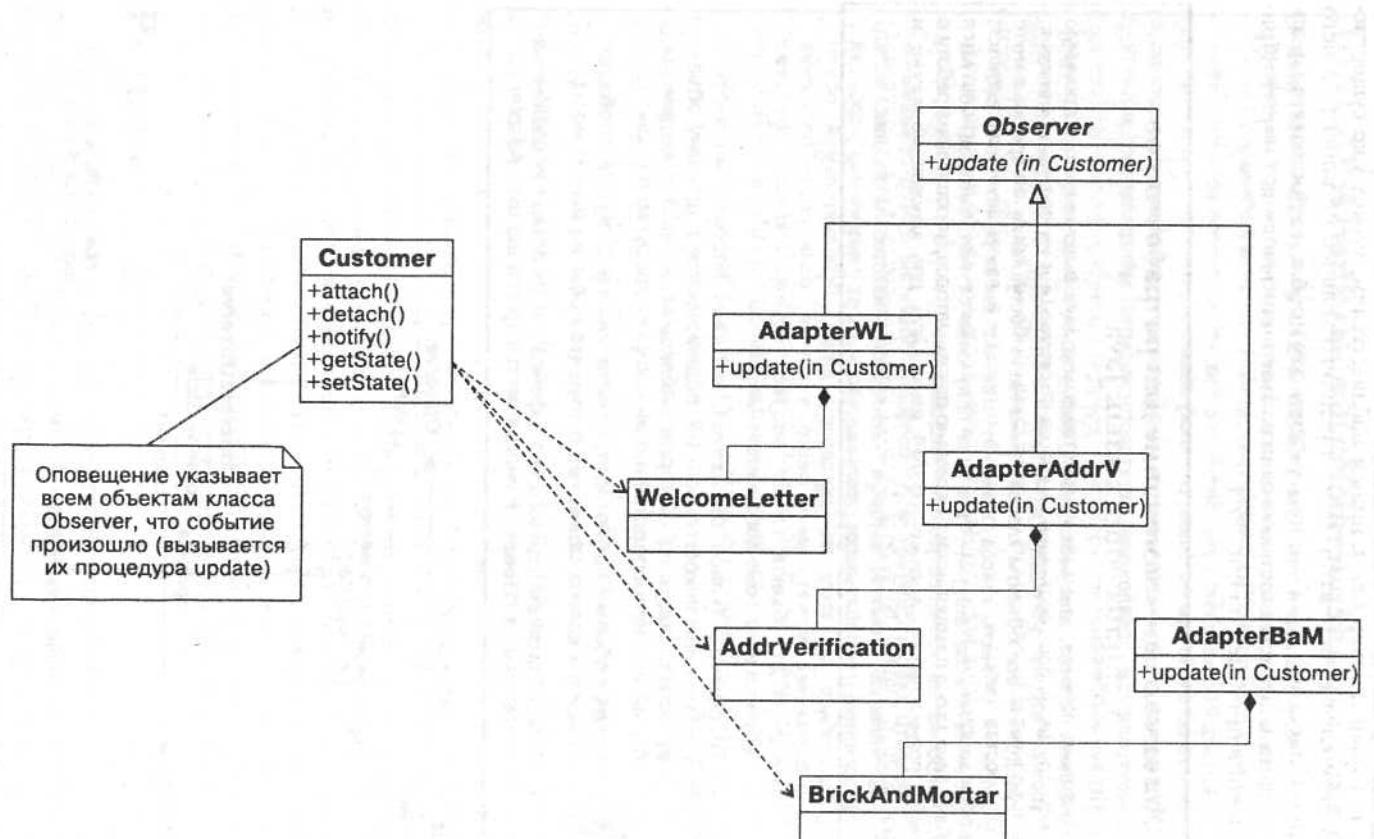


РИС. 17.4. Реализация классов-наблюдателей с помощью шаблона Adapter

Основные характеристики шаблона Observer

| | |
|----------------|--|
| Назначение | Определяет зависимость типа "один ко многим" между объектами таким образом, что когда состояние одного объекта изменяется, все зависящие от него объекты автоматически уведомляются об этом и обновляются |
| Задача | Необходимо направить уведомления о том, что некоторое событие имело место согласно изменяющемуся списку объектов |
| Способ решения | Объекты-наблюдатели (класс <i>Observer</i>) делегируют ответственность по контролю за появлением некоторого события центральному объекту (класс <i>Subject</i>) |
| Участники | Объект класса <i>Subject</i> знает всех своих объектов-наблюдателей, потому что они регистрируются у него. Объект класса <i>Subject</i> должен уведомить все объекты класса <i>Observer</i> о наступлении интересующего события. Объекты класса <i>Observer</i> отвечают как за регистрацию себя у объекта класса <i>Subject</i> , так и за получение от него необходимой информации после поступления уведомления |
| Следствия | Объекты класса <i>Subject</i> могут сообщить объектам класса <i>Observer</i> о наступлении событий, которые неинтересны некоторым из них, если последние обрабатывают только часть всех возможных событий (подробнее об этом мы поговорим ниже). Дополнительный обмен информацией может иметь место в случае, если после получения извещения от объекта класса <i>Subject</i> объектам класса <i>Observer</i> потребуются некоторые дополнительные данные |
| Реализация | <ul style="list-style-type: none"> • Те объекты (класс <i>Observer</i>), которым необходимо знать о наступлении некоего события, подключаются к другому объекту (класс <i>Subject</i>), который наблюдает за наступлением событий или непосредственно инициирует требуемое событие • Когда событие происходит, объект класса <i>Subject</i> сообщает объектам класса <i>Observer</i> о том, что событие имело место • Для реализации единого интерфейса у всех объектов различных типов класса <i>Observer</i> иногда используется шаблон Adapter |

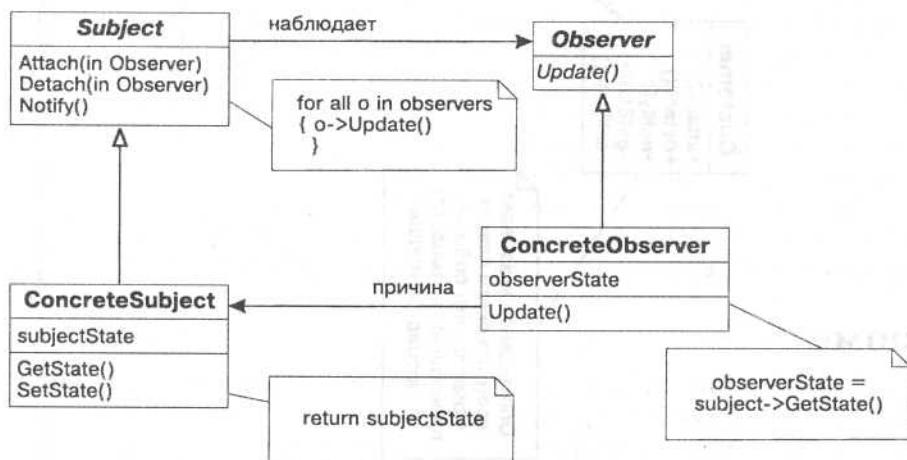


РИС. 17.5. Стандартный упрощенный вид шаблона Observer

Дополнительные замечания о шаблоне Observer

Следует заметить, что вовсе не обязательно использовать шаблон Observer всякий раз, когда между объектами возникает зависимость. Например, вполне очевидно, что в процессе подготовки накладной на заказ, объект, отвечающий за начисление налога, должен быть извещен каждый раз, когда в накладную добавляется новая строка, — чтобы сумма налога была пересчитана заново. Применение шаблона Observer в данном случае — это не лучший подход: последовательность операций известна заранее, и вряд ли что-либо когда-либо будет в нее добавлено. Когда зависимости между объектами фиксированы, применение шаблона Observer, скорее всего, только добавит в систему сложности.

Если список объектов, которые должны быть уведомлены о наступлении определенных событий, подвержен изменениям или зависит от конкретных условий, использование шаблона Observer, напротив, принесет большую пользу. Изменения в списке уведомляемых объектов могут быть вызваны как появлением новых требований к системе, так и динамическим появлением или удалением отдельных экземпляров объектов. Шаблон Observer также может быть полезен, если система будет функционировать в различных условиях или у различных заказчиков, каждый из которых имеет собственный список объектов-наблюдателей.

Определенные объекты-наблюдатели могут также обрабатывать только некоторые из возникающих событий. Вспомним пример с объектом **BrickandMortar**. В подобной ситуации объект-наблюдатель должен самостоятельно отфильтровать не интересующие его события.

Появление избыточных уведомлений может быть исключено посредством передачи ответственности за фильтрование рассылаемых уведомлений в адрес объекта класса **Subject**. Наилучший способ реализовать это решение — применить шаблон Strategy для проверки необходимости отправки уведомлений. В этом случае каждый объект-наблюдатель должен предоставить объекту класса **Subject** корректную стратегию принятия подобного решения непосредственно при своей регистрации.

Иногда объекты класса **Subject** могут вызывать метод `update` объектов-наблюдателей для передачи им информации. Такой подход помогает обойтись без обратных запросов от объектов-наблюдателей к объекту класса **Subject**. Однако часто оказывается, что разным объектам-наблюдателям требуется различная информация. В подобном случае вновь можно воспользоваться шаблоном Strategy. На этот раз объект класса **Strategy** используется для вызова метода `update` объектов-наблюдателей. И вновь объекты-наблюдатели должны предоставить для использования объекту класса **Subject** соответствующие объекты **Strategy**.

Резюме

Изучая шаблон Observer, мы определили, какой объект в системе позволит наилучшим образом справиться со вновь появляющимися изменениями. Для шаблона Observer объект, который инициирует событие (объект класса **Subject**), часто оказывается не в состоянии оповестить все объекты, которые должны быть извещены о наступлении этого события. Для решения проблемы создается интерфейс **Observer**

и устанавливается правило, согласно которому все объекты-наблюдатели обязаны зарегистрировать себя у объекта класса *Subject*.

В этой главе наше внимание было сосредоточено на шаблоне *Observer*, поэтому полезно будет указать несколько общих концепций объектно-ориентированного проектирования, используемых в этом шаблоне (табл. 17.3).

Таблица 17.3. Концепции ООП, используемые в шаблоне Observer

| Концепция | Пояснение |
|-------------------------------|---|
| Объекты отвечают сами за себя | Существуют различные виды объектов-наблюдателей, но все они получают необходимую им информацию от объекта класса <i>Subject</i> , а затем выполняют необходимые действия согласно их назначению |
| Абстрактные классы | Абстрактный класс <i>Observer</i> представляет концепцию объектов, которые нуждаются в получении уведомления. Он предоставляет классу <i>Subject</i> (субъекту) общий интерфейс для рассылки всех уведомлений |
| Полиморфизм и инкапсуляция | Субъект не знает, с каким именно типом объекта класса <i>Observer</i> он в данный момент взаимодействует. По существу, класс <i>Observer</i> инкапсулирует конкретные разновидности классов-наблюдателей. Это означает, что если в будущем появятся новые виды классов-наблюдателей, то это не потребует внесения каких-либо изменений в класс <i>Subject</i> |

Приложение. Пример программного кода на языке C++

Листинг 17.2. Реализация шаблона Observer

```
class Customer {
public:
    static void attach(Observer *o);
    static void detach(Observer *o);
    String getState();
private:
    Vector myObs;
    void notifyObs();
}
Customer::attach(Observer *o){
    myObs.addElement(o);
}
Customer::detach(Observer *o){
    myObs.remove(o);
}
Customer::getState () {
    // Могут использоваться другие методы,
    // предназначенные для получения дополнительной информации
}
Customer::notifyObs () {
    for (Enumeration e =
        myObs.elements();
        e.hasMoreElements() ;) {
```

```
    ((Observer *) e)->
        update(this);
    }
}

class Observer {
public:
    Observer();
    void update(Customer *mycust)=0;
    // Создает эту абстракцию
}
Observer::Observer () {
    Customer.attach( this );
}

class AddrVerification : public Observer {
public:
    AddrVerification();
    void update( Customer *myCust );
}
AddrVerification::AddrVerification () {
}
AddrVerification::update
(Customer *myCust) {
    // Здесь выполняется проверка адресов.
    // Для получения дополнительной информации о клиенте
    // может использоваться метод myCust
}

class WelcomeLetter : public Observer {
public:
    WelcomeLetter();
    void update( Customer *myCust );
}
WelcomeLetter::update( Customer *myCust ) {
    // Рассылка приветственного письма.
    // Для получения дополнительной информации о клиенте
    // может использоваться метод myCust
}
```

Глава 21

Шаблоны проектирования и новый взгляд на объектно- ориентированное проектирование

Введение

Завершая чтение книги, всегда полезно задать себе вопрос, что нового удалось из нее почерпнуть. В этой книге авторами была предпринята попытка дать читателю лучшее и, возможно, новое понимание принципов объектно-ориентированного проектирования, достигнутое за счет изучения шаблонов проектирования и разъяснения того, как эти шаблоны проектирования раскрывают объектно-ориентированную парадигму.

В этой главе мы выполним следующее.

- Познакомимся с новым взглядом на принципы объектно-ориентированного проектирования, базирующимся на понятии шаблонов проектирования.
- Выясним, как шаблоны проектирования могут помочь разработчику инкапсулировать реализацию.
- Обсудим метод анализа общности/изменчивости и выясним, как шаблоны проектирования помогают понять назначение абстрактных классов.
- Узнаем, как выполнить декомпозицию проблемной области на основе существующих в ней обязательств.
- Рассмотрим методы определения отношений между объектами.
- Еще раз вернемся к методу проектирования от контекста с использованием шаблонов проектирования.

В конце главы читателю будут предложены некоторые выводы и обобщения, основанные на практическом опыте авторов.

Перечень принципов объектно-ориентированного проектирования

В ходе обсуждения шаблонов проектирования упоминалось несколько важных принципов объектно-ориентированной парадигмы. Подводя итог, эти принципы можно сформулировать следующим образом.

- Объекты обладают четко определенными обязательствами.
- Объекты отвечают только за собственное поведение.
- Инкапсуляция подразумевает любой вид сокрытия, а именно:
 - сокрытие данных;
 - сокрытие класса (за абстрактным классом или интерфейсом);
 - сокрытие реализации.
- Выявление вариаций в поведении и данных с помощью анализа общности/изменчивости.
- Проектирование интерфейсов, а не реализаций.
- Понимание наследования как метода концептуализации вариаций, а не механизма определения особых версий уже существующих объектов.
- Исключение взаимосвязанности различных вариаций в одном и том же классе.
- Стремление к поддержанию низкой связанности в системе.
- Стремление к поддержанию высокой связности в объектах.
- Повсеместное соблюдение правила "однажды и только однажды" в отношении реализаций обязательств.

Как шаблоны проектирования инкапсулируют реализацию

Для некоторых из представленных в этой книге шаблонов проектирования характерно то, что они скрывают детали реализации от объекта-клиента. Например, шаблон *Bridge* скрывает от объекта-клиента подробности реализации классов, производных от класса *Abstraction*. Кроме того, интерфейс *Implementation* скрывает семейство классов реализации для класса *Abstraction* и его производных классов. В шаблоне *Strategy* от объекта-клиента скрыта реализация каждого класса *ConcreteStrategy*. Это правило справедливо для большинства шаблонов, описанных "бандой четырех", – все они позволяют скрыть конкретную реализацию.

Ценность подобного сокрытия реализации состоит в том, что благодаря ему шаблоны позволяют легко добавлять новые реализации, так как объекты-клиенты ничего не знают о том, как работают уже существующие реализации.

Анализ общности/изменчивости и шаблоны проектирования

В главе 9, *Шаблон Bridge*, было показано, как с помощью анализа общности/изменчивости можно вывести шаблон Bridge. Аналогичным образом могут быть выведены и многие другие шаблоны, включая Strategy, Iterator, Proxy, State, Visitor, Template Method и Abstract Factory. Однако более важно то, сколько шаблонов может быть *применено* за счет проведения анализа общности/изменчивости, поскольку поиск общностей в проблемной области помогает обнаружить присутствие в ней шаблонов.

Например, в отношении шаблона Bridge, можно начать анализ с рассмотрения нескольких конкретных требований к системе:

- вычерчивание квадрата с помощью первой графической программы;
- вычерчивание окружности с помощью второй графической программы;
- вычерчивание прямоугольника с помощью первой графической программы.

Знание шаблона Bridge позволяет выделить в этих конкретных случаях две общности:

- графические программы;
- отображаемые геометрические фигуры.

Аналогично, знание шаблона Strategy подсказывает, что если существует несколько различных правил, то следует искать в них возможную общность, которую затем можно будет инкапсулировать.

Но изучение шаблонов на этом не заканчивается. Необходимо продолжать читать литературу. Шаблонам посвящены различные дискуссии, проводимые на основании практического опыта анализа и проектирования. Шаблоны предоставляют команде разработчиков единый инструментарий для обсуждения проблемы, а также позволяют включить в создаваемый код лучшие практические решения, найденные другими программистами.

Декомпозиция проблемной области в обязательства

Анализ общности/изменчивости позволяет найти концептуальное решение (общность) и подготовить решение на уровне реализации (каждая конкретная вариация). Если учитывать только общности и объекты, использующие их, то можно пойти к решению проблемы с другой стороны – с помощью метода декомпозиции в обязательства.

Например, в шаблоне Bridge проблемная область рассматривается как состоящая из двух различных типов сущностей (абстракций и реализаций). Следовательно, не стоит ограничивать себя только объектно-ориентированной декомпозицией (т.е. разложением проблемной области на объекты), будет полезно также попробовать разложить проблемную область в обязательства, что может оказаться даже проще. В этом случае возможно предварительно определить объекты, которые потребуются для реализации найденных обязательств, и лишь затем переходить к обычной объектной декомпозиции.

Выше сказанное – это всего лишь расширение правила, которое я уже упоминал выше: проектировщик не должен беспокоиться о том, как будут создаваться экземпляры объектов, пока не станет известно, в каких именно объектах он нуждается. Это правило можно понимать как требование декомпозиции поставленной задачи на две части:

- определить, какие объекты необходимы;
- принять решение, как будут создаваться экземпляры этих объектов.

Те или иные шаблоны часто подсказывают нам, как можно выполнить декомпозицию обязательств. Например, шаблон Decorator демонстрирует, как обеспечить гибкое комбинирование объектов, если после декомпозиции проблемная область включает набор основных обязательств, которые используются всегда (класс **Concrete Component**), и некоторое множество вариаций, используемых от случая к случаю (классы-декораторы). Шаблон Strategy демонстрирует декомпозицию проблемы на объект, который использует правила, и собственно используемые правила.

Отношения внутри шаблона

Должен заметить, что на проводимых мной занятиях я иногда позволяю себе некоторую вольность, приводя определенную цитату из книги Александера. После того как две трети дня были посвящены обсуждению того, насколько хороши шаблоны проектирования, я беру в руки его книгу *Timeless Way of Building*, открываю последнюю страницу и говорю так.

В этой книге 549 страниц. На странице 545, которая, очевидно, одна из самых последних в книге, Александр говорит следующее: "На этой заключительной стадии шаблоны уже не важны..."¹

Я делаю паузу и замечаю: "Было бы прекрасно, если бы он сказал об этом в начале книги – это могло бы сэкономить нам уйму времени". Затем я продолжаю цитировать книгу: "Шаблоны учат нас быть восприимчивыми к реальности".²

И заканчиваю я такими словами: "Если вы прочтете книгу Александера, то поймете, что реальность – это отношения и движущие силы, описываемые шаблонами".

Шаблоны предоставляют нам способ говорить об этой реальности. Однако для нас важны вовсе не шаблоны сами по себе. Это же справедливо и по отношению к шаблонам проектирования в области разработки программного обеспечения.

Шаблон описывает движущие силы, мотивы и отношения для определенной проблемы в определенном контексте и предлагает подход к ее решению. Например, шаблон Bridge представляет отношения между классами, производными от определенной абстракции, и их возможными реализациями. Шаблон Strategy описывает отношения между следующими элементами:

- классом, который использует один из наборов алгоритмов (**Context**);
- членами этого набора алгоритмов (классы стратегий);

¹ Alexander C., Ishikawa S., Silverstein M. *The Timeless Way of Building*, NY: Oxford University Press, 1979, с. 545.

² Там же, с. 545.

- классом-клиентом, который использует класс **Context** и определяет, какой из алгоритмов следует использовать.

Шаблоны и проектирование от контекста

При обсуждении проблемы САПР в начале этой книги было показано, как шаблоны проектирования используются, если внимание сосредоточено на контексте, который они создают друг для друга. Шаблоны проектирования, функционирующие совместно, помогают найти оптимальную структуру приложения. Полезно будет указать, что многие шаблоны представляют собой типичные примеры проектирования от контекста.

Например:

- шаблон Bridge указывает на необходимость определять классы стороны реализации в контексте классов, производных от класса **Abstraction**;
- шаблон Decorator требует проектировать классы-декораторы в пределах контекста исходного компонента;
- шаблон Abstract Factory требует определять создаваемые семейства объектов в пределах контекста проблемы в целом, что позволяет установить, экземпляры каких именно классов должны быть реализованы.

Фактически, разработку с использованием интерфейсов и полиморфизма в общем случае можно рассматривать как один из видов проектирования по контексту. Взгляните на рис. 21.1, который представляет собой повторение рис. 8.4. Обратите внимание на то, что интерфейс абстрактного класса определяет тот контекст, в пределах которого должны быть реализованы все производные от него классы.

Анализируя, что именно эти объекты должны выполнять (концептуальное решение), мы определяем, как к ним следует обращаться (спецификации).



РИС. 21.1. Взаимосвязи между анализом общности/изменчивости, уровнями детализации и абстрактными классами

Дополнительные замечания

При изучении шаблонов проектирования будет весьма полезно рассмотреть следующие факторы и концепции.

- **Какие реализации этот шаблон скрывает?** Это позволит впоследствии изменять их.
- **Какие общности представлены в этом шаблоне?** Это поможет идентифицировать их.
- **Какими обязательствами обладают объекты в этом шаблоне?** Это может упростить выполнение декомпозиции обязательств.
- **Каковы отношения между заданными объектами?** Это предоставит информацию о движущих силах, представленных этими объектами.
- **Предоставляет ли сам шаблон некоторый пример проектирования по контексту?** Это позволит лучше понять, почему применение шаблонов является хорошим стилем проектирования.

Резюме

В этой главе подведен общий итог обсуждению нового подхода к объектно-ориентированному проектированию и показано, как шаблоны проектирования проявляются себя в этом. Рассматривая шаблоны проектирования, полезно ответить на вопросы, перечисленные ниже.

- Что инкапсулируют шаблоны проектирования?
- Как в них используется анализ общности/изменчивости?
- Как в таких шаблонах выполняется декомпозиция обязательств в проблемной области?
- Как они определяют отношения между объектами?
- Как они иллюстрируют проектирование по контексту?

Глава 22

Библиография

Эта книга – всего лишь введение. Введение в шаблоны проектирования, объектно-ориентированное проектирование и другие более мощные способы проектирования прикладных программных систем. Хотелось бы надеяться, что она обогатила вас определенными навыками, которые позволят вам реализовать на практике этот мощный и полезный стиль творческого мышления.

Каким может быть следующий этап в освоении идей, изложенных в этой книге? Чтобы помочь вам найти ответ на этот вопрос, в заключительной главе приведен аннотированный список источников, рекомендуемых к серьезному изучению.

В этой главе вы найдете следующее.

- Адреса Web-сайтов, дополняющих материал этой книги.
- Рекомендации для:
 - желающих углубить свои знания о шаблонах проектирования;
 - разработчиков на языке Java;
 - разработчиков на языке C++;
 - программистов на языке COBOL, которые хотят изучить объектно-ориентированную технологию;
 - всех желающих освоить новую мощную технологию разработки программного обеспечения, называемую XP (eXtreme Programming – экстремальное программирование).
- И, наконец, список книг, которые оказали влияние на меня лично и помогли понять, что жизнь – это нечто большее, чем программирование, и что гармоничное развитие личности поможет каждому стать действительно хорошим программистом.

Web-сайт поддержки данной книги

Web-сайт поддержки данной книги расположен по адресу:

<http://www.netobjectives.com/dpexplained>

На этом сайте можно найти такую дополнительную информацию о шаблонах проектирования.

- Примеры программного кода, ответы на часто задаваемые вопросы, материалы дискуссий, организованные по отдельным главам книги.
- Материалы дискуссий по проблемам рефакторинга.

- Сводные данные о шаблонах проектирования, представленные в удобном ссылочном формате.
- Описание курсов по изучению шаблонов проектирования и других близких тем.

Там же можно найти форму, чтобы отправить нам ваши комментарии и вопросы, касающиеся данной книги.

Кроме того, нами издается электронный журнал (*e-zine*), посвященный шаблонам проектирования и общим аспектам объектно-ориентированного проектирования. Для оформления подписки достаточно послать по электронной почте письмо с вашим именем и названием компании по адресу info@netobjectives.com со словом "subscribe" в поле *subject* (тема).

Рекомендуемая литература по шаблонам проектирования и объектно-ориентированной технологии

Я рекомендую к прочтению следующие книги и справочные руководства по объектно-ориентированному программированию и языку UML.

- Fowler M. *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison-Wesley, 2000.
Это наиболее полная известная мне работа о рефакторинге.
- Fowler M., Scott K. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2-nd Edition, Reading, MA: Addison-Wesley, 2000.
Этот источник я считаю лучшим для изучения языка UML. Данная книга будет полезной для начинающих, а также может использоваться как справочное руководство. Лично я пользуюсь ею постоянно.
- Meyer B. *Object-Oriented Software Construction*, Upper Saddle River, N.J.: Prentice Hall, 1997.
Невероятно насыщенная книга, написанная одним из наиболее выдающихся умов в нашей области.

Предмет, называемый "шаблонами проектирования", продолжает развиваться и углубляться. Изучать его можно на различных уровнях и со многих точек зрения. Я рекомендую следующие книги и справочные пособия, которые помогут вам на этом пути.

- Alexander C., Ishikawa S., Silverstein M. *The Timeless Way of Building*, New York, NY: Oxford University Press, 1979.
Это моя любимая книга как в профессиональном, так и в личном плане. Она одновременно интересна и глубока. Если вы прочитаете только одну книгу из данного списка, это должна быть именно она.
- Alexander C., Ishikawa S., Silverstein M. *A Pattern Language: Towns/Buildings/Construction*, New York, NY: Oxford University Press, 1977.
- Alexander C., Ishikawa S., Silverstein M. *Notes on Synthesis of Form*, New York, NY: Oxford University Press, 1970.

- Coplien J. *Multi-Paradigm Design for C++*, Reading, MA: Addison-Wesley, 1998.

Главы 2–5 этой книги следует прочитать даже тем, кто разрабатывает программное обеспечение на языке, отличном от C++. Данная книга описывает метод анализа общности/изменчивости лучше, чем какая-либо другая. На сайте нашей книги доступна интерактивная версия докторской диссертации Джима Коплина, которая представляет собой эквивалент этой книги.

- Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

Это издание продолжает оставаться лучшей из всех изданных книг о шаблонах проектирования. Ознакомление с ней является обязательным для каждого разработчика, работающего на языке C++.

- Gardner K. *Cognitive Patterns: Problem-Solving Frameworks for Object Technology*, New York, NY: Cambridge University Press, 1998.

Это взгляд на шаблоны проектирования с точки зрения науки о познании и искусственного интеллекта. На доктора Гарднера также оказала большое влияние книга Александера, приведенная первой в этом списке.

- Schmidt D., Stal M., Rohnert H., Busemann F. *Pattern-Oriented Software Architecture, Vol. 2*, New York, NY: John Wiley, 2000.

Книга об использовании шаблонов проектирования в многопоточных и распределенных средах.

- Vlissides J. *Pattern Hatching*, Reading, MA: Addison-Wesley, 1998.

Это отличная книга о шаблонах проектирования для подготовленных читателей. Иллюстрирует несколько способов организации совместной работы шаблонов. Предварительно я рекомендую прочитать данную книгу и книгу "банды четырех".

Рекомендуемая литература для программистов на языке Java

Когда я только начинал изучение языка Java, моими любимыми книгами были следующие.

- Eckel B. *Thinking in Java, 2nd Edition*, Upper Saddle River, N.J.: Prentice Hall, 2000.

Это одна из лучших книг о языке Java из числа изданных. Электронную версию этой книги можно найти по адресу <http://www.eckelobjects.com/DownloadSites>.

- Horstmann C. *Core Java 2, Volume 1, Fundamentals*, Palo Alto: Pearson Education, 1999. (Русский перевод этой книги Кея Хорстманна, *Java 2. Библиотека профессионала. Том 1. Основы*, готовится к выпуску издательским домом "Вильямс" в третьем квартале 2002 г.)

Еще одна хорошая книга для изучения основ языка Java.

Каждый язык программирования имеет собственный набор компонентов для реализации шаблонов проектирования. Если речь идет о языке Java, то я могу порекомендовать следующие книги.

- Coad P. *Java Design*, Upper Saddle River, N.J.: Prentice Hall, 2000.

Если вы считаете себя профессиональным разработчиком на языке Java, то я настоятельно рекомендую вам прочитать эту книгу. Здесь описывается большинство принципов и стратегий, которые будут весьма полезны при использовании шаблонов проектирования, несмотря на то, что собственно шаблоны в ней не упоминаются.

- Grand M. *Patterns in Java, Vol. 1*, New York, NY: John Wiley, 1998.

Если вы программируете на языке Java, эта книга также принесет вам большую пользу. В ней приведены интересные примеры программного кода и используется язык UML. Однако, по нашему мнению, обсуждение движущих сил и мотиваций в книге "банды четырех" более полезно, чем то, которое представлено в этой книге. Тем не менее, изучение отличного набора предлагаемых примеров представляет большую ценность, особенно для тех, кто практически работает с языком Java.

- Информацию об API языка Java для классов *Observer* и *Observable* можно найти по адресу <http://java.sun.com/j2se/1.3/docs/api/index.html>

Особого внимания в языке Java требует работа с потоками. Для углубленного изучения этой проблемы я рекомендую следующие издания.

- Hollub A. *Taming Java Threads*, Berkeley, CA: APress, 2000.
- Hyde P. *Java Thread Programming: The Authoritative Solution*, Indianapolis, IN: SAMS, 1999.
- Lea D. *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*, Reading, MA: Addison-Wesley, 2000.

Рекомендуемая литература для программистов на языке C++

Для тех, кто программирует на языке C++ в среде UNIX, я нахожу необходимым ознакомиться со следующей книгой.

- Stevens W. *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.

Это обязательный ресурс для каждого разработчика на языке C++ в среде UNIX.

Рекомендуемая литература для программистов на языке COBOL

Программистам, работающим с языком COBOL, которые хотят изучить объектно-ориентированное проектирование, можно порекомендовать следующий источник.

- Levey R. *Reengineering Cobol with Objects*, New York, NY: McGraw-Hill, 1995.

Полезная книга для программистов, работающих на языке COBOL и стремящихся освоить объектно-ориентированное проектирование.

Рекомендуемая литература для изучения технологии экстремального программирования

Для тех, кто хочет ознакомиться с технологией экстремального программирования (XP) или повысить свое мастерство в этой области, можно рекомендовать следующие два источника.

- <http://www.netobjectives.com/xp>

Наш собственный Web-сайт, посвященный экстремальному программированию и включающий статьи и курсы лекций по технологии XP.

- Beck K. *Extreme Programming Explained: Embrace Change*, Reading, MA: Addison-Wesley, 2000.

Это издание заслуживает внимания каждого, кто имеет отношение к разработке программного обеспечения, даже если он не планирует применять XP на практике. Я выбрал в этой книге 30 или около того страниц, содержащих, по моему мнению, важнейшие положения этой методологии, и поместил их список на нашем XP-сайте.

В настоящее время мы активно разрабатываем собственный метод проектирования программного обеспечения, который мы назвали Pattern-Accellerated Software Engineering. Он интегрирует несколько методов анализа и проектирования. Подробности можно найти по адресу <http://www.netobjectives.com/pase>.

Рекомендуемая литература по общим вопросам программирования

Данная книга отражает мою собственную философию, позволяет заглянуть в себя и увидеть, как каждому можно усовершенствовать самому и улучшить свою работу.

- Hunt A., Thomas D. *The Pragmatic Programmer: From Journeyman to Master*, Reading, MA: Addison-Wesley, 2000.

Это одна из тех прекрасных книг, которые я читаю по несколько страниц в день. Когда я встречаю упоминание о чем-то, что я уже использую, это укрепляет мою самооценку. Когда же я нахожу что-то новое, то получаю прекрасную возможность поучиться.

Наши любимые книги

Лично я полагаю, что лучший разработчик программного обеспечения – это не тот, кто живет и дышит одним лишь программированием. Пожалуй, именно способность думать и слушать, целостность и глубина личности, а также творческий подход – вот то, что, по моему мнению, присуще действительно хорошему разработчику. Такой человек более коммуникабелен. Он способен находить полезные идеи в других дисциплинах (мы, например, успешно применили в своей практике достижения из области архитектуры и антропологии). Системы, созданные такими разработчиками, ориентированы на людей, для которых эти системы, собственно, и предназначаются.

Многие студенты спрашивают нас о том, что мы любим читать, что оказало влияние на формирование наших взглядов и развитие наших личностей. Вот то, что можно было бы ответить на этот вопрос.

Алан рекомендует следующие издания.

- Grieve B. *The Blue Day Book: A Lesson in Cheering You Up*, Kansas City, KA: Andrews McMeel Publishing, 2000.

Это забавная и восхитительная книга. Обращайтесь к ней всякий раз, когда чувствуете себя не в своей тарелке.

- Hill N. *Think and Grow Rich*, New York, NY: Ballantine Books, 1960.

"Богатство" означает здесь не только деньги – данное слово обозначает все то, чем вы желаете обладать в своей жизни. Эта книга оказала большое влияние на мой личный успех и успех моего бизнеса.

- Kundtz D. *Stopping: How to Be Still When You Have to Keep Going*, Berkeley, CA: Conari Press, 1998.

Книга учит, как перестать быть трудоголиком. Она является прекрасным руководством в том, как отказаться от постоянного беспокойства и научиться наслаждаться жизнью, всегда добиваясь поставленных целей.

- Mandino O. *The Greatest Salesman in the World*, New York, NY: Bantam Press, 1968.

Я прочитал и применил положения из этой книги на практике несколько лет назад. Это помогло мне найти в жизни тот путь, который я всегда хотел найти. Если вы обратитесь к данной книге, я настоятельно рекомендую делать все то, о чем говорится в свитках Хафифа, а не ограничиваться лишь их прочтением (вы поймете, что я имею в виду, когда будете читать эту книгу).

- Pilzer P. *Unlimited Wealth: The Theory and Practice of Economic Alchemy*, Crown Publishers, 1990.

Эти книга представляет как новую парадигму ресурсов и источников благосостояния, так и рекомендации, как ими можно воспользоваться. Она будет крайне полезна каждому, кто живет в наш век информации.

- Remen R. *My Grandfather's Blessings: Stories of Strength, Refuge, and Belonging*, New York, NY: Riverhead Books, 2000.

Прекрасная книга, отражающая молитвы каждого достойного человека.

Джим рекомендует следующие издания.

- Buzan T., Buzan B. *The Mind Map Book: How to Use Radiant Thinking to Maximize Your Brain's Untapped Potential*, New York, NY: Dutton Books, 1994.
Эта книга оказала революционное влияние на мой стиль преподавания, общения с людьми, мышления и ведения записей. Невероятно мощная техника. Я пользуюсь ей ежедневно.
- Cahill T. *How the Irish Saved Civilization*, New York, NY: Doubleday, 1995.
Если в ваших жилах течет хоть немного ирландской крови, вы почувствуете гордость.
- Dawson C. *Religion and the Rise of Western Culture*, New York, NY: Doubleday, 1950.
Здесь описано, как религия направляла развитие западной цивилизации и способствовала удержанию в узде "варварства, которое скрывается внутри каждого из нас". Излагаются важные взгляды на научное мышление.
- Jensen B. *Simplicity: The New Competitive Advantage in a World of More, Better, Faster*, Cambridge, MA: Perseus Books, 2000.
Революция в мышлении и управлении знаниями. Системы проектирования будут проще в использовании для людей, если особенности человека были приняты во внимание при разработке процессов и технологий.
- Lingenfelter S. *Transforming Culture*, Grand Rapids: Baker Book House, 1998.
Модель для понимания особенностей культур через теорию социальных игр.
- Spradely J. P. *The Ethnographic Interview*, New York, NY: Harcourt Brace Jovanovich College Publishers, 1979.
Эту книгу следует прочесть каждому, кто хочет научиться брать интервью. Классический учебник, знакомый всем студентам, изучающим антропологию.
- Wiig K. *Knowledge Management Methods*, Dallas, AL: Schema Press, 1995.
Виртуальная энциклопедия методов, помогающих организациям более эффективно использовать информационные ресурсы.