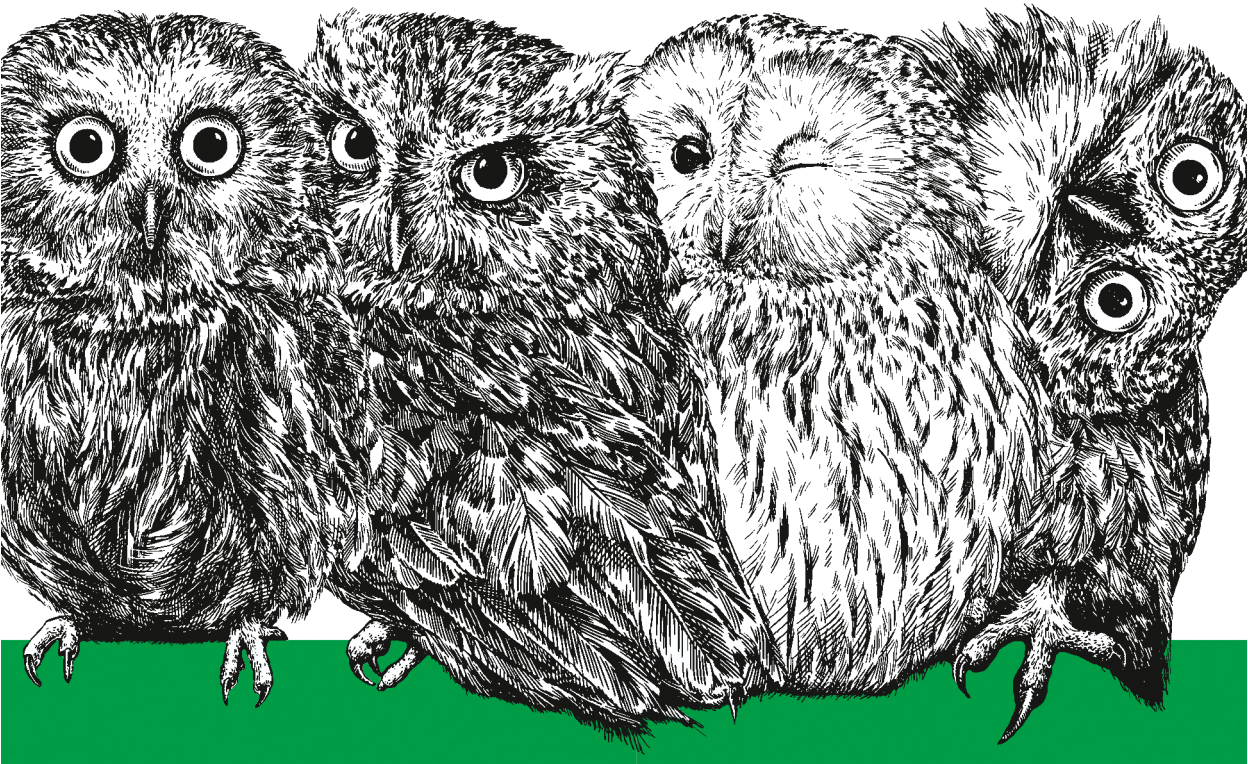
БИБЛИОТЕКА ПРОГРАММИСТА

ЮБИЛЕЙНОЕ ИЗДАНИЕ
ЛЕГЕНДАРНОЙ КНИГИ
БАНДЫ ЧЕТЫРЕХ



ПАТТЕРНЫ
ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Э. Гамма, Р. Хелм. Р. Джонсон, Дж. Влиссидес

Erich Gamma, Richard Helm, Ralph Johnson,
John Vlissides

Design Patterns.

Elements of Reusable Object-Oriented Software

Addison-Wesley

An imprint of Addison Wesley Longman, Inc.
Reading, Masachusetts • Harlow, England • Menlo Park, California
Berkley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Э. Гамма, Р. Хелм,
Р. Джонсон, Дж. Влиссидес

ПАТТЕРНЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

ЮБИЛЕЙНОЕ ИЗДАНИЕ ЛЕГЕНДАРНОЙ КНИГИ БАНДЫ ЧЕТЫРЕХ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2021

ББК 32.973.2-018-02
УДК 004.43
П75

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.

П75 Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021. — 448 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1595-2

Больше 25 лет прошло с момента выхода первого тиража книги Design Patterns. За это время книга из популярной превратилась в культовую. Во всем мире ее рекомендуют прочитать каждому, кто хочет связать жизнь с информационными технологиями и программированием. «Русский» язык, на котором разговаривают айтишники, поменялся, многие англоязычные термины стали привычными, паттерны вошли в нашу жизнь.

Перед вами юбилейное издание с обновленным переводом книги, ставшей must-read для каждого программиста. «Паттерны объектно-ориентированного проектирования» пришли на смену «Приемам объектно-ориентированного проектирования».

Четыре первоклассных разработчика — Банда четырех — представляют вашему вниманию опыт ООП в виде двадцати трех паттернов. Паттерны появились потому, что разработчики искали пути повышения гибкости и степени повторного использования своих программ. Авторы не только дают принципы использования шаблонов проектирования, но и систематизируют информацию. Вы узнаете о роли паттернов в архитектуре сложных систем и сможете быстро и эффективно создавать собственные приложения с учетом всех ограничений, возникающих при разработке больших проектов. Все шаблоны взяты из реальных систем и основаны на реальной практике. Для каждого паттерна приведен код на C++ или Smalltalk, демонстрирующий его возможности.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.43

ISBN 978-0201633610 англ.
ISBN 978-5-4461-1595-2

КРАТКОЕ СОДЕРЖАНИЕ

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Книга не является введением в объектно-ориентированное программирование или проектирование. На эти темы написано много других хороших книг. Предполагается, что вы достаточно хорошо владеете по крайней мере одним объектно-ориентированным языком программирования и имеете какой-то опыт объектно-ориентированного проектирования. Безусловно, у вас не должно возникать необходимости лезть в словарь за разъяснением терминов «тип», «полиморфизм», и вам понятно, чем «наследование интерфейса» отличается от «наследования реализации».

С другой стороны, эта книга и не научный труд, адресованный исключительно узким специалистам. Здесь говорится о *паттернах проектирования* и описываются простые и элегантные решения типичных задач, возникающих в объектно-ориентированном проектировании. Паттерны проектирования не появились сразу в готовом виде; многие разработчики, искавшие возможности повысить гибкость и степень пригодности к повторному использованию своих программ, приложили много усилий, чтобы поставленная цель была достигнута. В паттернах проектирования найденные решения воплощены в краткой и легко применимой на практике форме.

Для использования паттернов не нужны ни какие-то особенные возможности языка программирования, ни хитроумные приемы, поражающие воображение друзей и начальников. Все можно реализовать на стандартных объектно-ориентированных языках, хотя для этого потребуется приложить несколько больше усилий, чем в случае специализированного решения, применимого только в одной ситуации. Но эти усилия неизменно окупаются за счет большей гибкости и возможности повторного использования.

Когда вы усвоите работу с паттернами проектирования настолько, что после удачного их применения воскликнете «Ага!», а не будете смотреть в сомне-

нии на получившийся результат, ваш взгляд на объектно-ориентированное проектирование изменится раз и навсегда. Вы сможете строить более гибкие, модульные, повторно используемые и понятные конструкции, а разве не для этого вообще существует объектно-ориентированное проектирование?

Несколько слов, чтобы предупредить и одновременно подбодрить вас. Не огорчайтесь, если не все будет понятно после первого прочтения книги. Мы и сами не всё понимали, когда начинали писать ее! Помните, что эта книга не из тех, которые, однажды прочитав, ставят на полку. Надеемся, что вы будете возвращаться к ней снова и снова, черпая идеи и ожидая вдохновения.

Книга созревала довольно долго. Она повидала четыре страны, была свидетелем женитьбы трех ее авторов и рождения двух младенцев. В ее создании так или иначе участвовали многие люди. Особую благодарность мы выражаем Брюсу Андерсону (Bruce Anderson), Кенту Беку (Kent Beck) и Андре Вейнанду (Andre Weinand) за поддержку и ценные советы. Также благодарим всех рецензентов черновых вариантов рукописи: Роджера Билефельда (Roger Bielefeld), Грейди Буча (Grady Booch), Тома Каргилла (Tom Cargill), Маршалла Клайна (Marshall Cline), Ральфа Хайра (Ralph Hyre), Брайана Кернигана (Brian Kernighan), Томаса Лалиберти (Thomas Laliberty), Марка Лоренца (Mark Lorenz), Артура Риля (Arthur Riel), Дуга Шмидта (Doug Schmidt), Кловиса Тондо (Clovis Tondo), Стива Виноски (Steve Vinoski) и Ребекку Вирфс-Брок (Rebecca Wirfs-Brock). Выражаем признательность сотрудникам издательства AddisonWesley за поддержку и терпение: Кейту Хабибу (Kate Habib), Тиффани Мур (Tiffany Moore), Лайзе Раффаэле (Lisa Raffaele), Прадипе Сива (Pradeepa Siva) и Джону Уэйту (John Wait). Особая благодарность Карлу Кесслеру (Carl Kessler), Дэнни Саббаху (Danny Sabbah) и Марку Вегману (Mark Wegman) из исследовательского отдела компании IBM за неослабевающий интерес к этой работе и поддержку.

И наконец, не в последнюю очередь мы благодарны всем тем людям, которые высказывали замечания по поводу этой книги по интернету, ободряли нас и убеждали, что такая работа действительно нужна. Вот далеко не полный перечень наших «незнакомых помощников»: Йон Авотинс (Jon Avotins), Стив Берчук (Steve Berczuk), Джулиан Бердич (Julian Berdych), Матиас Болен (Matthias Bohlen), Джон Брант (John Brant), Алан Кларк (Allan Clarke), Пол Чизхолм (Paul Chisholm), Йенс Колдьюи (Jens Coldewey), Дейв Коллинз (Dave Collins), Джим Коплиен (Jim Coplien), Дон Двиггинс (Don Dwiggins), Габриэль Элиа (Gabriele Elia), Дуг Фельт (Doug Felt), Брайан Фут (Brian Foote), Денис Фортин (Denis Fortin), Уорд Харольд (Ward Harold), Херман Хуэни (Hermann Hueni), Найим Ислам (Nayeem Islam), Бикрамжит Калра (Bikramjit Kalra), Пол Кифер (Paul Keefer),

Томас Кофлер (Thomas Kofler), Дуг Леа (Doug Lea), Дэн Лалиберте (Dan LaLiberte), Джеймс Лонг (James Long), Анна Луиза Луу (Ann Louise Luu), Панди Мадхаван (Pundi Madhavan), Брайан Мэрик (Brian Marick), Роберт Мартин (Robert Martin), Дэйв МакКомб (Dave McComb), Карл МакКоннелл (Carl McConnell), Кристин Мингинс (Christine Mingins), Ханспетер Мессенбек (Hanspeter Mossenbock), Эрик Ньютон (Eric Newton), Марианна Озкан (Marianne Ozkan), Роксана Пайетт (Roxsan Payette), Ларри Подмолик (Larry Podmolik), Джордж Радин (George Radin), Сита Рамакришнан (Sita Ramakrishnan), Русс Рамирес (Russ Ramirez), Александр Ран (Alexander Ran), Дирк Риле (Dirk Riehle), Брайан Розенбург (Bryan Rosenburg), Аамод Сейн (Aamod Sane), Дури Шмидт (Duri Schmidt), Роберт Зайдль (Robert Seidl), Цинь Шу (Xin Shu) и Билл Уокер (Bill Walker).

Мы не считаем, что набор отобранных нами паттернов полон и неизменен, он всего лишь отражает наши нынешние представления о проектировании. Мы приветствуем любые замечания, будь то критика приведенных примеров, ссылки на известные способы использования, которые не упомянуты здесь, или предложения по поводу дополнительных паттернов. Вы можете писать нам на адрес издательства Addison-Wesley или на электронный адрес *design-patterns@cs.uiuc.edu*. Исходные тексты всех примеров можно получить, отправив сообщение «send design pattern source» по адресу *design-patterns-source@cs.uiuc.edu*. А теперь также есть веб-страница *http://st-www.cs.uiuc.edu/users/patterns/DPBook/DPBook.html*, на которой размещается последняя информация и обновления к книге.

Эрих Гамма	Маунтин Вью, штат Калифорния
Ричард Хелм	Монреаль, Квебек
Ральф Джонсон	Урбана, штат Иллинойс
Джон Влиссидес	Готорн, штат Нью-Йорк

Август 1994

ОТ ИЗДАТЕЛЬСТВА

С момента издания классической книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (Design Patterns: Elements of Reusable Object-Oriented Software) прошло 26 лет. За это время было продано более полумиллиона экземпляров книги на английском и 13 других языках. На этой книге выросло не одно поколение программистов.

В книге описываются простые и изящные решения типичных задач, возникающих в объектно-ориентированном проектировании.

Паттерны появились потому, что многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме. «Банда Четырех» объясняет каждый паттерн на простом примере четким и понятным языком. Использование паттернов при разработке программных систем позволяет проектировщику перейти на более высокий уровень разработки проекта. Теперь архитектор и программист могут оперировать образными названиями паттернов и общаться на одном языке.

Таким образом, книга решает две задачи.

Во-первых, знакомит с ролью паттернов в создании архитектуры сложных систем.

Во-вторых, позволяет проектировщикам с легкостью разрабатывать собственные приложения, применяя содержащиеся в справочнике паттерны.

Что изменилось в издании 2020 года?

- Актуализирована терминология (например, для «реорганизации» кода уже вполне прижился термин «рефакторинг», для share — «совместное использование» вместо «разделения», а для mixin — «примесь»);
- обновлен стиль;
- устранены излишне громоздкие слова (например, «специфицирование» или «инстанцирование». Первое можно вполне адекватно заменить «определением», второе — «созданием экземпляра»);
- книга наконец-то называется «Паттерны объектно-ориентированного проектирования».

В квадратных скобках даются ссылки на источники (см. Библиографию), а цифры в круглых скобках обозначают ссылку на страницу, где описывается тот или иной паттерн.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

ВВЕДЕНИЕ В ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Проектирование объектно-ориентированных программ — нелегкое дело, а если они предназначены для *повторного использования*, то все становится еще сложнее. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить ключевые отношения между классами. Дизайн должен, с одной стороны, соответствовать решаемой задаче, с другой — быть общим, чтобы удалось учесть все требования, которые могут возникнуть в будущем. Хотелось бы также избежать вовсе или, по крайней мере, свести к минимуму необходимость перепроектирования. Поднаторевшие в объектно-ориентированном проектировании разработчики скажут вам, что создать «правильный», то есть в достаточной мере гибкий и пригодный для повторного использования дизайн, с первой попытки очень трудно, если вообще возможно. Прежде чем считать цель достигнутой, они обычно пытаются опробовать найденное решение на нескольких задачах, и каждый раз модифицируют его.

И все же опытным проектировщикам удается создать хороший дизайн системы. В то же время новички испытывают шок от количества возможных вариантов и нередко возвращаются к привычным не объектно-ориентированным методикам. Проходит немало времени перед тем, как новички поймут, что же такое удачный объектно-ориентированный дизайн. Очевидно, опытные проектировщики знают какие-то тонкости, ускользающие от новичков. Так что же это?

Прежде всего, опытный разработчик понимает, что *не нужно* решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться

теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение один раз, он будет прибегать к нему снова и снова. Именно благодаря накопленному опыту проектировщик и становится экспертом в своей области. Во многих объектно-ориентированных системах встречаются повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированная архитектура становится более гибкой, элегантной, и может использоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед.

Поясним нашу мысль через аналогию. Писатели редко выдумывают совершенно новые сюжеты. Вместо этого они берут за основу уже отработанные в мировой литературе схемы, жанры и образы. Например, персонаж — «трагический герой» (Макбет, Гамлет и т. д.), жанр — «любовный роман» (бесчисленные любовные романы). Точно так же в объектно-ориентированном проектировании используются такие паттерны, как «представление состояния с помощью объектов» или «декорирование объектов, чтобы было проще добавлять и удалять новую функциональность». Если вы знаете паттерн, многие проектировочные решения далее следуют автоматически.

Все мы знаем о ценности опыта. Сколько раз при проектировании вы испытывали дежавю, чувствуя, что уже когда-то решали такую же задачу, только никак не сообразить, когда и где? Если бы удалось вспомнить детали старой задачи и ее решения, то не пришлось бы придумывать все заново. Увы, у нас нет привычки записывать свой опыт на благо другим людям да и себе тоже.

Цель этой книги состоит как раз в том, чтобы документировать опыт разработки объектно-ориентированных программ в виде паттернов проектирования. Каждому паттерну мы присвоим имя, объясним его назначение и роль в проектировании объектно-ориентированных систем. Мы хотели отразить опыт проектирования в форме, которую другие люди могли бы использовать эффективно. Для этого некоторые из наиболее распространенных паттернов были формализованы и сведены в единый каталог.

Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования делает их более доступными для разработчиков новых систем. Паттерны проектирования помогают выбрать альтернативные решения, упрощающие повторное использование системы, и избежать тех альтернатив, которые его затрудняют. Паттерны улучшают качество документации и сопровождения существующих систем,

поскольку они позволяют явно описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе. Проще говоря, паттерны проектирования дают разработчику возможность быстрее найти правильный путь.

Ни один из паттернов, представленных в книге, не описывает новые или непроверенные разработки. В книгу были включены только такие паттерны, которые неоднократно применялись в разных системах. По большей части они никогда ранее не документировались и либо хорошо известны только в объектно-ориентированном сообществе, либо были частью какой-то удачной объектно-ориентированной системы — ни один источник нельзя назвать простым для начинающих проектировщиков. Таким образом, хотя эти решения не новы, мы представили их в новом и доступном формате: в виде каталога паттернов в едином формате.

Хотя книга получилась довольно объемной, паттерны проектирования — лишь малая часть того, что необходимо знать специалисту в этой области. В издание не включено описание паттернов, имеющих отношение к параллелизму, распределенному программированию и программированию систем реального времени. Отсутствуют и сведения о паттернах, специфичных для конкретных предметных областей. Из этой книги вы не узнаете, как строить интерфейсы пользователя, как писать драйверы устройств и как работать с объектно-ориентированными базами данных. В каждой из этих областей есть свои собственные паттерны; возможно, в будущем кто-то систематизирует и их.

1.1. ЧТО ТАКОЕ ПАТТЕРН ПРОЕКТИРОВАНИЯ

По словам Кристофера Александера (Christopher Alexander), «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, и при этом никакие две реализации не будут полностью одинаковыми» [AIS+77]. Хотя Александер имел в виду паттерны, возникающие при проектировании зданий и городов, но его слова верны и в отношении паттернов объектно-ориентированного проектирования. Наши решения выражаются в терминах объектов и интерфейсов, а не стен и дверей, но в обоих случаях смысл паттерна — предложить решение определенной задачи в конкретном контексте.

В общем случае паттерн состоит из четырех основных элементов:

1. **Имя.** Указывая имя, мы сразу описываем проблему проектирования, ее решения и их последствия — и все это в одном-двух словах. Присваивание

паттернам имен расширяет наш «словарный запас» проектирования и позволяет проектировать на более высоком уровне абстракции. Наличие словаря паттернов позволяет обсуждать их с коллегами, в документации и даже с самим собой. Имена позволяют анализировать дизайн системы, обсуждать его достоинства и недостатки с другими. Нахождение хороших имен было одной из самых трудных задач при составлении каталога.

2. **Задача.** Описание того, когда следует применять паттерн. Описание объясняет задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. В нем могут быть отмечены структуры классов или объектов, типичные для негибкого дизайна. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.

3. **Решение.** Описание элементов дизайна, отношений между ними, их обязанностей и взаимодействий между ними. В решении не описывается конкретный дизайн или реализация, поскольку паттерн — это шаблон, применимый в самых разных ситуациях. Вместо этого дается абстрактное описание задачи проектирования и ее возможного решения с помощью некоего обобщенного сочетания элементов (в нашем случае классов и объектов).

4. **Результаты** — следствия применения паттерна, его вероятные плюсы и минусы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна. Нередко к результатам относится баланс затрат времени и памяти, а также речь может идти о выборе языка и подробностях реализации. Поскольку в объектно-ориентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы. Перечисление всех последствий поможет вам понять и оценить их роль.

Вопрос о том, что считать паттерном, а что нет, зависит от точки зрения. То, что один воспринимает как паттерн, для другого просто примитивный строительный блок. В этой книге мы рассматриваем паттерны на определенном уровне абстракции. *Паттерны проектирования* — это не то же самое, что связанные списки или хештаблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. С другой стороны, это и не сложные предметно-ориентированные решения для целого приложения или подсистемы. *В этой книге под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте*.

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он выделяет участвующие классы и экземпляры, их роли и отношения, а также распределение обязанностей. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. В формулировке паттерна анализируется, когда следует применять паттерн, можно ли его использовать с учетом других проектных ограничений, каковы будут последствия применения метода. Поскольку любой проект в конечном итоге предстоит реализовывать, в состав паттерна включается пример кода на языке C++ (иногда на Smalltalk), иллюстрирующего реализацию.

Хотя в паттернах описываются объектно-ориентированные архитектуры, они основаны на практических решениях, реализованных на основных языках объектно-ориентированного программирования типа Smalltalk и C++, а не на процедурных (Pascal, C, Ada и т. п.) или более динамических объектно-ориентированных языках (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из прагматических соображений, поскольку наш опыт повседневного программирования связан именно с этими языками, и они завоевывают все большую популярность.

Выбор языка программирования безусловно важен. В наших паттернах подразумевается использование возможностей Smalltalk и C++, и от этого зависит, что реализовать легко, а что — трудно. Если бы мы ориентировались на процедурные языки, то включили бы паттерны наследование, инкапсуляция и полиморфизм. Некоторые из наших паттернов напрямую поддерживаются менее распространенными языками. Так, в языке CLOS есть мультиметоды, которые делают ненужным паттерн посетитель (с. 379). Собственно, даже между Smalltalk и C++ есть много различий, из-за чего некоторые паттерны проще выражаются на одном языке, чем на другом (см., например, паттерн итератор — с. 302).

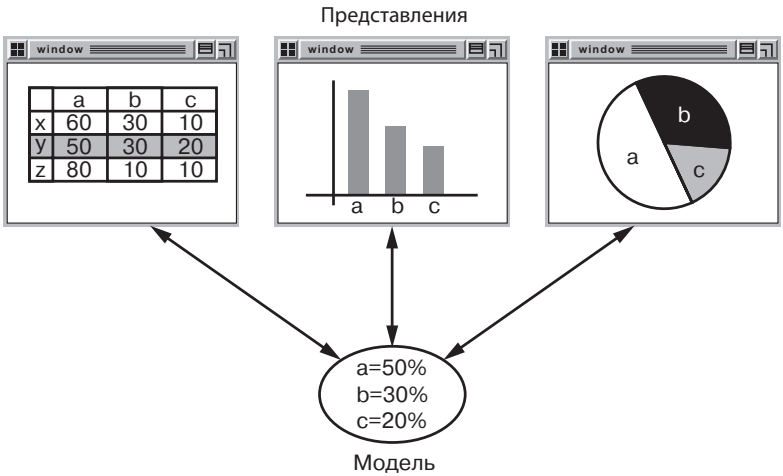
1.2. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В СХЕМЕ MVC В ЯЗЫКЕ SMALLTALK

В Smalltalk80 для построения интерфейсов пользователя применяется тройка классов модель/представление/контроллер (Model/View/Controller — MVC) [KP88]. Знакомство с паттернами проектирования, встречающимися в схеме MVC, поможет вам разобраться в том, что мы понимаем под словом «паттерн».

MVC состоит из объектов трех видов. *Модель* — это объект приложения, а *представление* — его внешний вид на экране. *Контроллер* описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования.

MVC отделяет представление от модели, устанавливая между ними протокол взаимодействия «подписка/уведомление». Представление должно гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель уведомляет все зависящие от нее представления, в результате чего представление обновляет себя. Такой подход позволяет присоединить к одной модели несколько представлений, обеспечив тем самым различные представления. Можно создать новое представление, не переписывая модель.

На следующей схеме показана одна модель и три представления. (Для простоты мы опустили контроллеры.) Модель содержит некоторые данные, которые могут быть представлены в форме электронной таблицы, гистограммы и круговой диаграммы. Модель сообщает своим представлениям обо всех изменениях значений данных, а представления взаимодействуют с моделью для получения новых значений.



На первый взгляд, в этом примере продемонстрирован просто дизайн, отделяющий представление от модели. Но тот же принцип применим и к более общей задаче: разделение объектов таким образом, что изменение одного

отражается сразу на нескольких других, причем изменившийся объект не имеет информации о подробностях реализации других объектов. Этот более общий подход описывается паттерном проектирования наблюдатель.

Еще одна особенность MVC заключается в том, что представления могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составное представление, содержащее вложенные — по одной кнопке на каждое. Пользовательский интерфейс инспектора объектов может состоять из вложенных представлений, используемых также и в отладчике. MVC поддерживает вложенные представления с помощью класса `CompositeView`, являющегося подклассом `View`. Объекты класса `CompositeView` ведут себя так же, как объекты класса `View`, поэтому могут использоваться всюду, где и представления. Но еще они могут содержать вложенные представления и управлять ими.

Здесь можно было бы считать, что этот дизайн позволяет обращаться с составным представлением, как с любым из его компонентов. Но тот же дизайн применим и в ситуации, когда мы хотим иметь возможность группировать объекты и рассматривать группу как отдельный объект. Такой подход описывается паттерном компоновщик. Он позволяет создавать иерархию классов, в которой некоторые подклассы определяют примитивные объекты (например, `Button` — кнопка), а другие — составные объекты (`CompositeView`), группирующие примитивы в более сложные структуры.

MVC позволяет также изменять реакцию представления на действия пользователя. При этом визуальное воплощение остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать открывающиеся меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте `Controller`. Существует иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Представление пользуется экземпляром класса, производного от `Controller`, для реализации конкретной стратегии реагирования. Чтобы реализовать иную стратегию, нужно просто подставить другой контроллер. Можно даже заменить контроллер представления во время выполнения программы, изменив тем самым реакцию на действия пользователя. Например, представление можно деактивировать, так что он вообще не будет ни на что реагировать, если передать ему контроллер, игнорирующий события ввода.

Отношение представление/контроллер — это пример паттерна проектирования стратегия (с. 362). Стратегия — это объект, представляющий алгоритм. Он будет полезен, когда вы хотите статически или динамически подменить

один алгоритм другим, если существует много разновидностей одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать.

В MVC используются и другие паттерны проектирования, например фабричный метод (с. 135), позволяющий задать для представления класс контроллера по умолчанию, и декоратор (с. 209) для добавления к представлению возможности прокрутки. Тем не менее, основные отношения в схеме MVC описываются паттернами наблюдатель, компоновщик и стратегия.

1.3. ОПИСАНИЕ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Как мы будем описывать паттерны проектирования? Графические обозначения важны, но их одних недостаточно. Они просто символизируют конечный продукт процесса проектирования в виде отношений между классами и объектами. Чтобы повторно воспользоваться дизайном, нам необходимо документировать решения, альтернативные варианты и компромиссы, которые привели к нему. Важны также конкретные примеры, поскольку они демонстрируют практическое применение паттерна.

При описании паттернов проектирования мы будем придерживаться единого формата. Описание каждого паттерна разбито на разделы, перечисленные ниже. Такой подход позволяет единообразно представить информацию, облегчает изучение, сравнение и применение паттернов.

Название и классификация паттерна

Название паттерна должно быть компактным и четко отражающим его назначение. Выбор названия чрезвычайно важен, потому что оно станет частью вашего словаря проектирования. Классификация паттернов проводится в соответствии со схемой, которая изложена в разделе 1.5.

Назначение

Краткие ответы на следующие вопросы: что делает паттерн? Почему и для чего он был создан? Какую конкретную задачу проектирования можно решить с его помощью?

Другие названия

Другие распространенные названия паттерна, если таковые имеются.

Мотивация

Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.

Применимость

Описание ситуаций, в которых можно применять данный паттерн. Примеры неудачного проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций.

Структура

Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT) [RBP+91]. Мы пользуемся также диаграммами взаимодействий [JCJO92, Boo94] для иллюстрации последовательностей запросов и отношений между объектами. В приложении Б эта нотация описывается подробно.

Участники

Классы или объекты, задействованные в данном паттерне проектирования, и их обязанности.

Отношения

Взаимодействие участников для выполнения своих обязанностей.

Результаты

Насколько паттерн удовлетворяет поставленным требованиям? К каким результатам приводит паттерн, на какие компромиссы приходится идти? Какие аспекты структуры системы можно независимо изменять при использовании данного паттерна?

Реализация

О каких сложностях и подводных камнях следует помнить при реализации паттерна? Существуют ли какие-либо советы и рекомендуемые приемы? Есть ли у данного паттерна зависимость от языка программирования?

Пример кода

Фрагменты кода, демонстрирующие возможную реализацию на языках C++ или Smalltalk.

Известные применения

Возможности применения паттерна в реальных системах. Приводятся по меньшей мере два примера из различных областей.

Родственные паттерны

Какие паттерны проектирования тесно связаны с данным? Какие важные различия существуют между ними? С какими другими паттернами хорошо сочетается данный паттерн?

В приложениях приводится общая информация, которая поможет вам лучше понять паттерны и связанные с ними вопросы. Приложение А содержит глоссарий употребляемых нами терминов. В уже упомянутом приложении Б дано описание разнообразных нотаций. Некоторые аспекты применяемой нотации мы поясняем по мере ее появления в тексте книги. Наконец, в приложении В приведен исходный код фундаментальных классов, встречающихся в примерах.

1.4. КАТАЛОГ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Каталог, начинающийся на с. 108, содержит 23 паттерна. Ниже для удобства перечислены их имена и назначение. В скобках после названия каждого паттерна указан номер страницы, откуда начинается его подробное описание.

Abstract Factory (абстрактная фабрика) (113)

Предоставляет интерфейс для создания семейств связанных между собой или зависимых объектов без указания их конкретных классов.

Adapter (адаптер) (171)

Преобразует интерфейс класса в другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.

Bridge (мост) (184)

Отделяет абстракцию от реализации, чтобы их можно было изменять независимо друг от друга.

ГЛАВА 2

ПРАКТИЧЕСКИЙ ПРИМЕР: ПРОЕКТИРОВАНИЕ РЕДАКТОРА ДОКУМЕНТОВ

В данной главе рассматривается применение паттернов на примере проектирования визуального редактора документов Lexi[1], построенного по принципу «что видишь, то и получаешь» (WYSIWYG). Вы увидите, как с помощью паттернов решаются проблемы проектирования, характерные для Lexi и аналогичных приложений. К концу этой главы у вас появится практический опыт использования восьми паттернов.

На рис. 2.1 изображен пользовательский интерфейс редактора Lexi. WYSIWYG-представление документа занимает большую прямоугольную область в центре. В документе могут произвольно сочетаться текст и графика с применением разных стилей форматирования. Вокруг документа — привычные выпадающие меню и полосы прокрутки, а также значки с номерами для перехода на нужную страницу документа.

2.1. ЗАДАЧИ ПРОЕКТИРОВАНИЯ

Рассмотрим семь задач, характерных для дизайна Lexi.

- **Структура документа.** Выбор внутреннего представления документа отражается практически на всех аспектах дизайна. Для редактирования, форматирования, отображения и анализа текста необходимо

[1] Дизайн Lexi основан на программе Doc — текстовом редакторе, разработанном Кальдером [CL92].

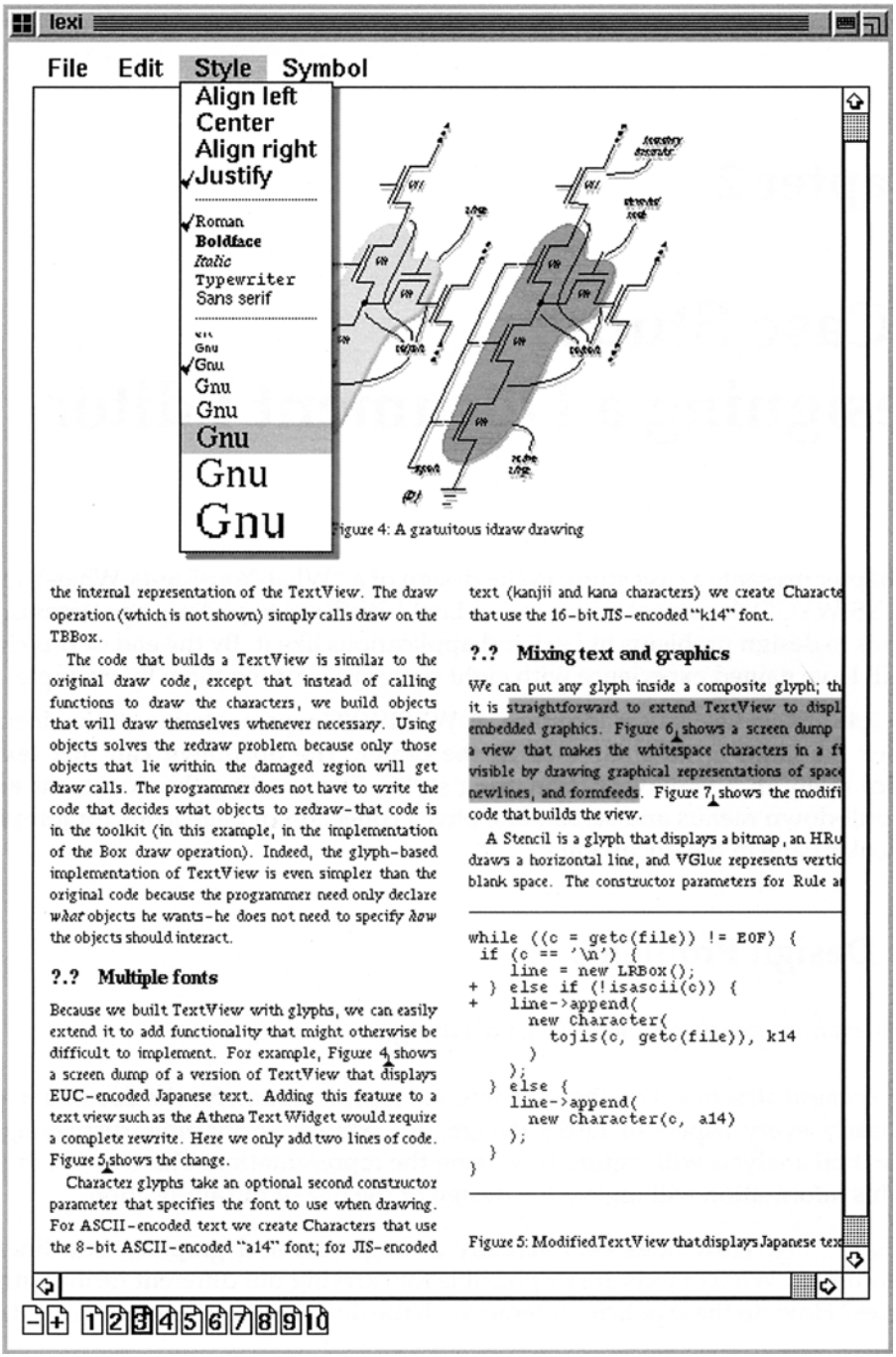


Рис. 2.1. Пользовательский интерфейс Lexi

уметь перебирать составляющие этого представления. Способ организации информации играет решающую роль при дальнейшем проектировании.

- **Форматирование.** Как в Lexi организовано размещение текста и графики в виде ряда колонок? Какие объекты отвечают за реализацию различных политик форматирования? Как эти политики взаимодействуют с внутренним представлением документа?
- **Создание привлекательного интерфейса пользователя.** В состав пользовательского интерфейса Lexi входят полосы прокрутки, рамки и эффекты тени у выпадающих меню. Вполне вероятно, что эти украшения будут изменяться по мере развития интерфейса Lexi. Поэтому важно иметь возможность легко добавлять и удалять элементы оформления, не затрагивая приложение.
- **Поддержка разных стандартов оформления программы.** Lexi должен без серьезной модификации адаптироваться к стандартам оформления программ, например, таким как Motif или Presentation Manager (PM).
- **Поддержка оконных систем.** В разных оконных системах обычно используются разные стандарты оформления и поведения. Дизайн Lexi должен по возможности быть независимым от оконной системы.
- **Операции пользователя.** Пользователи управляют работой Lexi с помощью элементов интерфейса, в том числе кнопок и выпадающих меню. Функциональность, которая вызывается из интерфейса, разбросана по многим объектам программы. Проблема в том, чтобы разработать единообразный механизм для обращения к таким функциям и отмены уже выполненных операций.
- **Проверка правописания и расстановка переносов.** Поддержка в Lexi таких аналитических операций, как проверка правописания и определение мест расстановки переносов. Как свести к минимуму число классов, которые придется модифицировать при добавлении новой аналитической операции?

Ниже обсуждаются указанные проблемы проектирования. Для каждой из них определяются некоторые цели и ограничения на способы их достижения. Прежде чем предлагать решение, мы подробно остановимся на целях и ограничениях. На примере проблемы и ее решения демонстрируется применение одного или нескольких паттернов проектирования. Обсуждение каждой проблемы завершается краткой характеристикой паттерна.

2.2. СТРУКТУРА ДОКУМЕНТА

Документ — это всего лишь организованное некоторым способом множество базовых графических элементов: символов, линий, многоугольников и других геометрических фигур. В совокупности они образуют полную информацию о содержании документа. И все же создатель документа часто представляет себе эти элементы не в графическом виде, а в терминах физической структуры документа — строк, колонок, рисунков, таблиц и других подструктур[1]. Эти подструктуры, в свою очередь, составлены из более мелких и т. д.

Пользовательский интерфейс Lexi должен позволять пользователям работать с такими подструктурами напрямую. Например, пользователю следует предоставить возможности, которые позволят ему обращаться с диаграммой как с неделимой единицей, а не как с набором отдельных графических примитивов; с таблицей — как с единым целым, а не как с неструктурированным хранилищем текста и графики. Это делает интерфейс простым и интуитивно понятным. Чтобы реализация Lexi обладала аналогичными свойствами, мы выберем внутреннее представление, соответствующее физической структуре документа.

В частности, внутреннее представление должно поддерживать:

- отслеживание физической структуры документа, то есть разбиение текста и графики на строки, колонки, таблицы и т. д.;
- генерирование визуального представления документа;
- установление соответствия между позициями экрана и элементами внутреннего представления. Это позволит определить, что имеет в виду пользователь, выбирая некоторый элемент визуального представления.

Кроме целей, также имеются и ограничения. Во-первых, текст и графику следует трактовать единообразно. Интерфейс приложения должен позволять свободно размещать текст внутрь графики и наоборот. Не следует считать графику частным случаем текста или текст — частным случаем графики, поскольку это в конечном итоге приведет к появлению избыточных механизмов форматирования и манипулирования. Одного набора механизмов должно быть достаточно и для текста, и для графики.

---

[1] Авторы часто рассматривают документы и в терминах их *логической* структуры: предложений, абзацев, разделов, подразделов и глав. Чтобы не усложнять пример, мы не будем явно хранить во внутреннем представлении информацию о логической структуре. Но то проектное решение, которое мы опишем, вполне пригодно для представления и такой информации.

Во-вторых, в нашей реализации не может быть различий во внутреннем представлении отдельного элемента и группы элементов. Если Lexi будет одинаково работать с простыми и составными элементами, это позволит создавать документы со структурой любой сложности. Например, десятым элементом в строке второй колонки может быть как один символ, так и сложно устроенная диаграмма со многими внутренними компонентами. Если вы уверены в том, что этот элемент умеет изображать себя на экране и сообщать свои размеры, его внутренняя сложность не имеет никакого отношения к тому, как и в каком месте страницы он отображается.

Однако второе ограничение противоречит необходимости анализировать текст на предмет выявления орфографических ошибок и расстановки переносов. Во многих случаях нам безразлично, является ли элемент строки простым или сложным объектом. Но иногда анализ зависит от анализируемого объекта. Так, вряд ли имеет смысл проверять орфографию многоугольника или пытаться переносить его с одной строки на другую. При проектировании внутреннего представления надо учитывать эти и другие ограничения, которые могут конфликтовать друг с другом.

РЕКУРСИВНАЯ КОМПОЗИЦИЯ

На практике для представления информации, имеющей иерархическую структуру, часто применяется прием, называемый *рекурсивной композицией*. Он позволяет строить все более сложные элементы из простых. Рекурсивная композиция дает возможность составить документ из простых графических элементов. Сначала мы можем линейно расположить множество символов и графики слева направо для формирования одной строки документа. Затем несколько строк можно объединить в колонку, несколько колонок — в страницу и т. д. (рис. 2.2).

Для представления физической структуры можно ввести отдельный объект для каждого существенного элемента. К таковым относятся не только видимые элементы вроде символов и графики, но и структурные элементы — строки и колонки. В результате получается структура объекта, изображенная на рис. 2.3.

Представляя объектом каждый символ и каждый графический элемент документа, мы обеспечиваем гибкость на самых нижних уровнях дизайна Lexi. Текст и графика обрабатываются единообразно в том, что касается отображения, форматирования и вложения друг в друга. Lexi можно расширить для поддержки новых наборов символов, не затрагивая никаких других функций. Объектная структура Lexi точно отражает физическую структуру документа.

ГЛАВА 3

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

Порождающие паттерны проектирования абстрагируют процесс создания экземпляров. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать класс создаваемого экземпляра, а паттерн, порождающий объекты, делегирует создание экземпляров другому объекту. Эти паттерны начинают играть более важную роль, когда система эволюционирует и начинает в большей степени зависеть от композиции объектов, чем от наследования классов. При этом основной акцент смещается с жесткого кодирования фиксированного набора поведений на определение небольшого набора фундаментальных поведений, посредством композиции которых можно получить любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое создание экземпляра класса.

Для порождающих паттернов характерны два аспекта. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, они скрывают подробности создания и компоновки экземпляров этих классов. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость в отношении того, *что* создается, *кто* это создает, *как* и *когда*. Это позволяет настроить систему «готовыми» объектами с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

В некоторых ситуациях возможен выбор между тем или иным порождающим паттерном. Например, есть случаи, когда с пользой для дела можно

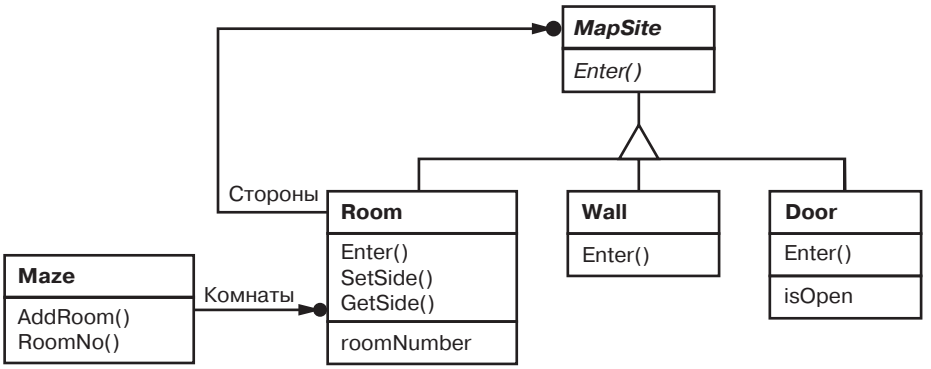
использовать как прототип (146), так и абстрактную фабрику (113). В других ситуациях порождающие паттерны дополняют друг друга. Так, применяя паттерн строитель (124), можно использовать другие паттерны для решения вопроса о том, какие компоненты нужно строить, а прототип (146) может использовать одиночку (157) в своей реализации.

Поскольку порождающие паттерны тесно связаны друг с другом, мы изучим сразу все пять, чтобы лучше были видны их сходства и различия. Изучение будет вестись на общем примере — построении лабиринта для компьютерной игры. Правда, и лабиринт, и игра будут слегка варьироваться для разных паттернов. Иногда целью игры станет просто отыскание выхода из лабиринта; тогда игроку будет доступно только локальное представление лабиринта. В других случаях в лабиринтах могут встречаться задачи, которые игрок должен решить, и опасности, которые ему предстоит преодолеть. В подобных играх может отображаться карта того участка лабиринта, который уже был исследован.

Мы опустим многие детали того, что может встречаться в лабиринте, и предназначен ли лабиринт для одного или нескольких игроков, а сосредоточимся лишь на принципах создания лабиринта. Лабиринт будет определяться как множество комнат. Любая комната «знает» о своих соседях, в качестве которых могут выступать другая комната, стена или дверь в другую комнату.

Классы `Room` (комната), `Door` (дверь) и `Wall` (стена) определяют компоненты лабиринта и используются во всех наших примерах. Мы определим только те части этих классов, которые важны для создания лабиринта. Не будем рассматривать игроков, операции отображения и блуждания в лабиринте и другие важные функции, не имеющие отношения к построению лабиринта.

На схеме ниже показаны отношения между классами `Room`, `Door` и `Wall`.



У каждой комнаты есть четыре стороны. Для задания северной, южной, восточной и западной сторон будем использовать перечисление `Direction` в реализации на языке C++:

```
enum Direction {North, South, East, West};
```

В программах на языке Smalltalk для представления направлений будут использоваться соответствующие символические имена.

Класс `MapSite` — общий абстрактный класс для всех компонентов лабиринта. Для упрощения примера в нем определяется только одна операция `Enter`, смысл которой зависит от того, куда именно вы входите. Когда вы входите в комнату, ваше местоположение изменяется. При попытке затем войти в дверь может произойти одно из двух. Если дверь открыта, то вы попадаете в следующую комнату, а если закрыта, то вы разбиваете себе нос:

```
class MapSite {
public:
    virtual void Enter() = 0;
};
```

Операция `Enter` составляет основу для более сложных игровых операций. Например, если вы находитесь в комнате и говорите «Иду на восток», то игрой определяется, какой объект класса `MapSite` находится к востоку от вас, и для него вызывается операция `Enter`. Определенные в подклассах операции `Enter` «выяснят», изменили вы свое местоположение или расшибли нос. В реальной игре `Enter` мог бы передаваться аргумент с объектом, представляющим блуждающего игрока.

`Room` — это конкретный подкласс класса `MapSite`, который определяет ключевые отношения между компонентами лабиринта. Он содержит ссылки на другие объекты `MapSite`, а также хранит номер комнаты. Все комнаты в лабиринте идентифицируются номерами:

```
class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

Следующие классы представляют стены и двери, находящиеся с каждой стороны комнаты:

```
class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

Тем не менее, информации об отдельных частях лабиринта недостаточно. Определим еще класс `Maze` для представления набора комнат. В этот класс включена операция `RoomNo` для нахождения комнаты по ее номеру:

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    // ...
};
```

`RoomNo` могла бы выполнять поиск с помощью линейного списка, хеш-таблицы или даже простого массива. Но пока нас не интересуют такие подробности. Займемся тем, как описать компоненты объекта, представляющего лабиринт.

Определим также класс `MazeGame`, который создает лабиринт. Самый простой способ сделать это — строить лабиринт последовательностью операций, добавляющих к нему компоненты, которые потом соединяются. Например, следующая функция создаст лабиринт из двух комнат с одной дверью между ними:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Функция получилась довольно сложной, если учесть, что она всего лишь строит лабиринт из двух комнат. Есть очевидные способы упростить ее. Например, конструктор класса `Room` мог бы инициализировать стороны без дверей заранее; но это означает лишь перемещение кода в другое место. Суть проблемы не в размере этой функции, а в ее *негибкости*. Структура лабиринта жестко «зашита» в функции. Чтобы изменить структуру, придется изменить саму функцию, либо заместив ее (то есть полностью переписав заново), либо непосредственно модифицируя ее фрагменты. Оба пути чреваты ошибками и не способствуют повторному использованию.

Порождающие паттерны показывают, как сделать дизайн более *гибким*, хотя и необязательно меньшим по размеру. В частности, их применение позволит легко менять классы, определяющие компоненты лабиринта.

Предположим, вы хотите использовать уже существующую структуру в новой игре с волшебными лабиринтами. В такой игре появляются не существовавшие ранее компоненты, например `DoorNeedingSpell` — запертая дверь, для открывания которой нужно произнести заклинание, или `EnchantedRoom` — комната, где есть необычные предметы, скажем, волшебные ключи или магические слова. Как легко изменить операцию `CreateMaze`, чтобы она создавала лабиринты с новыми классами объектов?

В данном случае самое серьезное препятствие лежит в жестко зашитой информации о классах, экземпляры которых создаются в коде. С помощью

ГЛАВА 4

СТРУКТУРНЫЕ ПАТТЕРНЫ

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня *класса* используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Этот паттерн особенно полезен для организации совместной работы нескольких независимо разработанных библиотек. Другой пример паттерна уровня класса — адаптер (171). В общем случае адаптер делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

Вместо композиции интерфейсов или реализаций структурные паттерны уровня *объекта* компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Примером структурного паттерна уровня объектов является компоновщик (196). Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать структуры произвольной сложности из примитивных и других составных объектов. В паттерне заместитель (246) объект берет на себя функции другого объекта.

У заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве; может представлять большой объект, загружаемый по требованию, или ограничивать доступ к критически важному объекту. Заместитель вводит дополнительный косвенный уровень доступа к отдельным свойствам объекта. Поэтому он может ограничивать, расширять или изменять эти свойства.

Паттерн приспособленец (231) определяет структуру для совместного использования объектов. Владельцы совместно используют объекты, по меньшей мере, по двум причинам: для достижения эффективности и непротиворечивости. Приспособленец акцентирует внимание на эффективности использования памяти. В приложениях, в которых участвует очень много объектов, должны снижаться накладные расходы на хранение. Значительной экономии можно добиться за счет разделения объектов вместо их дублирования. Но объект может быть разделяемым, только если его состояние не зависит от контекста. У объектов-приспособленцев такой зависимости нет. Любая дополнительная информация передается им по мере необходимости. В отсутствие состояния, зависящего от контекста, объекты-приспособленцы могут легко использоваться совместно.

Если паттерн приспособленец дает способ работы с большим числом мелких объектов, то паттерн фасад (221) показывает, как один объект может представлять целую подсистему. Фасад представляет набор объектов и выполняет свои функции, перенаправляя сообщения объектам, которые он представляет. Паттерн мост (184) отделяет абстракцию объекта от его реализации, так что их можно изменять независимо.

Паттерн декоратор (209) описывает динамическое добавление объектам новых обязанностей. Это структурный паттерн, который рекурсивно компонует объекты с целью реализации заранее неизвестного числа дополнительных функций. Например, объект-декоратор, содержащий некоторый элемент пользовательского интерфейса, может добавить к нему оформление в виде рамки или тени либо новую функциональность, например возможность прокрутки или изменения масштаба. Два разных оформления прибавляются путем простого вложения одного декоратора в другой. Для достижения этой цели каждый объект-декоратор должен соблюдать интерфейс своего компонента и перенаправлять ему сообщения. Свои функции (скажем, рисование рамки вокруг компонента) декоратор может выполнять как до, так и после перенаправления сообщения.

Многие структурные паттерны в той или иной мере связаны друг с другом. Эти отношения обсуждаются в конце главы.

ПАТТЕРН ADAPTER (АДАПТЕР)

Название и классификация паттерна

Адаптер — паттерн, структурирующий классы и объекты.

Назначение

Преобразует интерфейс одного класса в другой интерфейс, на который рассчитаны клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Другие названия

`Wrapper` (обертка).

Мотивация

Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удается использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

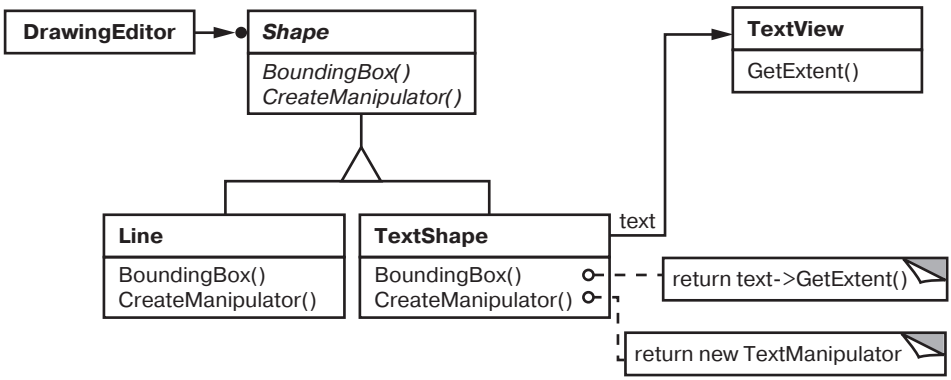
Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т. д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, форма которого может редактироваться пользователем и который умеет выполнять прорисовку самого себя. Интерфейс графических объектов определен абстрактным классом `Shape`. Редактор определяет подкласс класса `Shape` для каждого вида графических объектов: `LineShape` для прямых, `PolygonShape` для многоугольников и т. д.

Классы для элементарных геометрических фигур, например `LineShape` и `PolygonShape`, реализуются сравнительно просто, поскольку заложенные в них возможности рисования и редактирования ограничены по своей природе. Но подкласс `TextShape`, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет хорошо проработанный класс `TextView`, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать `TextView` для реали-

зации `TextShape`, но библиотека разрабатывалась без учета классов `Shape`, поэтому использовать `TextView` вместо `Shape` не удастся.

Так каким же образом существующие и независимо разработанные классы вроде `TextView` могут работать в приложении, спроектированном под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса `TextView`, чтобы он соответствовал интерфейсу `Shape`, но для этого понадобится исходный код. Впрочем, если он доступен, то вряд ли будет разумно изменять `TextView`; библиотека не должна приспосабливаться к интерфейсам каждого конкретного приложения только для того, чтобы приложение заработало.

Вместо этого можно было бы определить класс `TextShape` так, что он будет *адаптировать* интерфейс `TextView` к интерфейсу `Shape`. Это можно сделать двумя способами: (1) наследованием интерфейса от `Shape`, а реализации от `TextView`; (2) включением экземпляра `TextView` в `TextShape` и реализацией `TextShape` в категориях интерфейса `TextView`. Два данных подхода соответствуют вариантам паттерна `адаптер` в его классовой и объектной ипостасях. Класс `TextShape` мы будем называть *адаптером*.



На этой схеме показан адаптер объекта. Видно, как запрос `BoundingBox`, объявленный в классе `Shape`, преобразуется в запрос `GetExtent`, определенный в классе `TextView`. Поскольку класс `TextShape` адаптирует `TextView` к интерфейсу `Shape`, графический редактор может воспользоваться классом `TextView`, хотя тот и имеет несовместимый интерфейс.

Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс. На схеме показано, как адаптер реализует такого рода обязанности. У пользователя должна быть возможность перемещать

любой объект класса `Shape` в другое место, но в классе `TextView` такая операция не предусмотрена. `TextShape` может добавить недостающую функциональность, самостоятельно реализовав операцию `CreateManipulator` класса `Shape`, которая возвращает экземпляр подходящего подкласса `Manipulator`.

`Manipulator` — это абстрактный класс для объектов, которые умеют анимировать `Shape` в ответ на такие действия пользователя, как перетаскивание фигуры в другое место. У класса `Manipulator` имеются подклассы для различных фигур. Например, `TextManipulator` — подкласс для `TextShape`. Возвращая экземпляр `TextManipulator`, объект класса `TextShape` добавляет новую функциональность, которой в классе `TextView` нет, а классу `Shape` требуется.

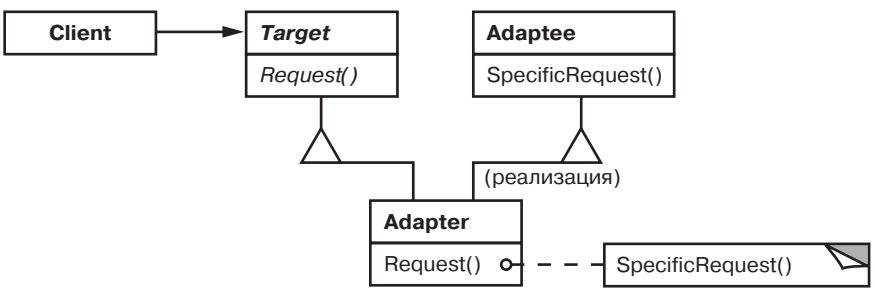
Применимость

Основные условия для применения паттерна адаптер:

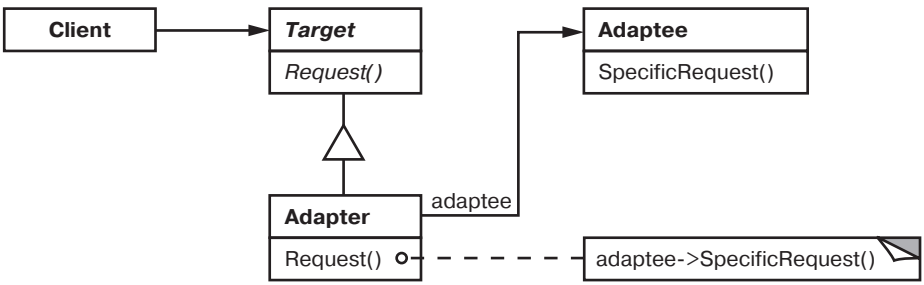
- вы хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- требуется создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (*только для адаптера объектов*!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



Адаптер объекта применяет композицию объектов.



Участники

- **Target** (`Shape`) — целевой:
  - определяет зависящий от предметной области интерфейс, которым пользуется `Client`;
- **Client** (`DrawingEditor`) — клиент:
  - вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу `Target`;
- **Adaptee** (`TextView`) — адаптируемый:
  - определяет существующий интерфейс, который нуждается в адаптации;
- **Adapter** (`TextShape`) — адаптер:
  - адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Отношения

Клиенты вызывают операции экземпляра адаптера `Adapter`. В свою очередь адаптер вызывает операции адаптируемого объекта или класса `Adaptee`, который и выполняет запрос.

Результаты

Адаптеры объектов и классов обладают разными достоинствами и недостатками. Адаптер класса:

- адаптирует `Adaptee` к `Target`, перепоручая действия конкретному классу `Adaptee`. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;

ГЛАВА 5

ПАТТЕРНЫ ПОВЕДЕНИЯ

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных схемах взаимодействия между ними. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентируется не на схеме управления как таковой, а на связях между объектами.

В паттернах поведения уровня класса для распределения поведения между разными классами используется наследование. В этой главе описано два таких паттерна. Из них более простым и широко распространенным является шаблонный метод (373), который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм детализируется за счет подклассов, где определяются абстрактные операции. Другой паттерн поведения уровня класса — интерпретатор (287) — представляет грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над экземплярами этих классов.

В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Одноранговые объекты могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Эту проблему решает паттерн посредник (319). Посредник, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрыва лишних связей.

Паттерн цепочка обязанностей (263) позволяет и дальше уменьшать степень связанности. Он дает возможность посылать запросы объекту не напрямую, а по цепочке «объектов-кандидатов». Запрос может выполнить любой «кандидат», если это допустимо в текущем состоянии выполнения программы. Число кандидатов заранее не определено, а подбирать участников можно во время выполнения.

Паттерн наблюдатель (339) определяет и поддерживает зависимости между объектами. Классический пример наблюдателя встречается в схеме «модель —представление — контроллер» языка Smalltalk, где все виды модели уведомляются о любых изменениях ее состояния.

Прочие паттерны поведения связаны с инкапсуляцией поведения в объекте и делегированием ему запросов. Паттерн стратегия (362) инкапсулирует алгоритм объекта, упрощая его спецификацию и замену. Паттерн команда (275) инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн состояние (352) инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять свое поведение. Паттерн посетитель (379) инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн итератор (302) абстрагирует механизм доступа и обхода объектов из некоторого агрегата.

ПАТТЕРН CHAIN OF RESPONSIBILITY (ЦЕПОЧКА ОБЯЗАННОСТЕЙ)

Название и классификация паттерна

Цепочка обязанностей — паттерн поведения объектов.

Назначение

Позволяет избежать привязки отправителя запроса к его получателю, предоставляя возможность обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос по этой цепочке, пока он не будет обработан.

Мотивация

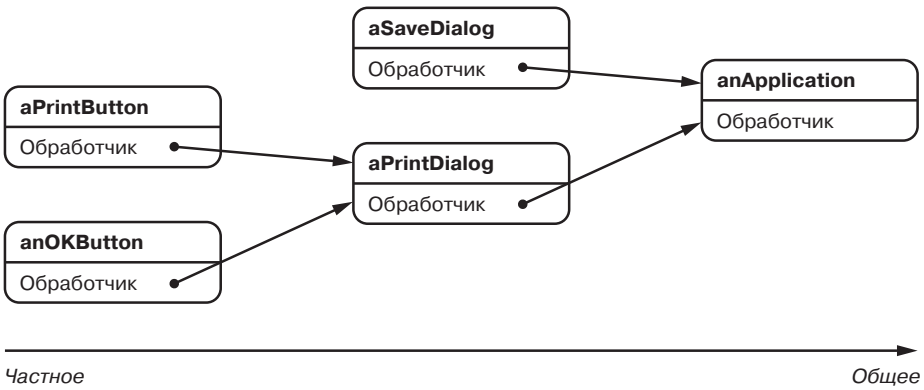
Рассмотрим контекстнозависимую оперативную справку в графическом интерфейсе: пользователь может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание

справки зависит от того, какая часть интерфейса была выбрана и в каком контексте. Например, справка по кнопке в диалоговом окне может отличаться от справки по аналогичной кнопке в главном окне приложения. Если для некоторой части интерфейса справки нет, то система должна показать информацию о ближайшем контексте, в котором она находится — например, о диалоговом окне в целом.

Следовательно, естественно было бы организовать справочную информацию от более конкретных разделов к более общим. Кроме того, ясно, что запрос на получение справки обрабатывается одним из нескольких объектов пользовательского интерфейса, а каким именно — зависит от контекста и имеющейся в наличии информации.

Проблема в том, что объект, *инициирующий* запрос (например, кнопка), не знает, какой объект в конечном итоге предоставит справку. Необходимо каким-то образом отделить кнопку — инициатор запроса от объектов, которые могут предоставить справочную информацию. Паттерн цепочка обязанностей показывает, как это может происходить.

Идея паттерна заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока не будет обработан одним из них.



Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который действует точно так же. У объекта, отправившего запрос, отсутствует информация об обработчике. Мы говорим, что у запроса есть *анонимный получатель* (implicit receiver).

Допустим, пользователь запрашивает справку по кнопке Print (печать). Она находится в диалоговом окне `PrintDialog`, содержащем информацию об объ-

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес

Паттерны объектно-ориентированного проектирования

Перевел с английского А. Слинкин

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*Е. Матвеев*
Художественный редактор	*В. Мостипан*
Корректор	*М. Молчанова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 08.09.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 36,120. Доп. тираж 2000. Заказ 0000.