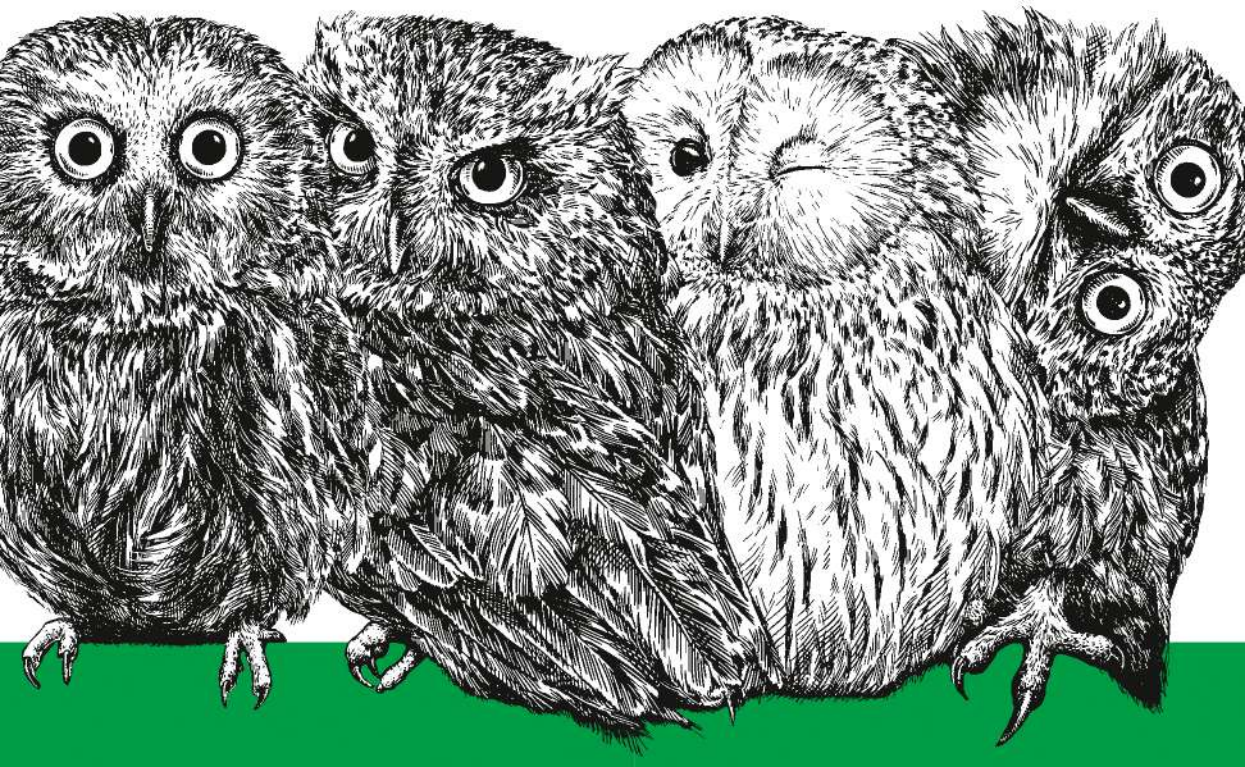
БИБЛИОТЕКА ПРОГРАММИСТА

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

2020

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, 2020. — 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).

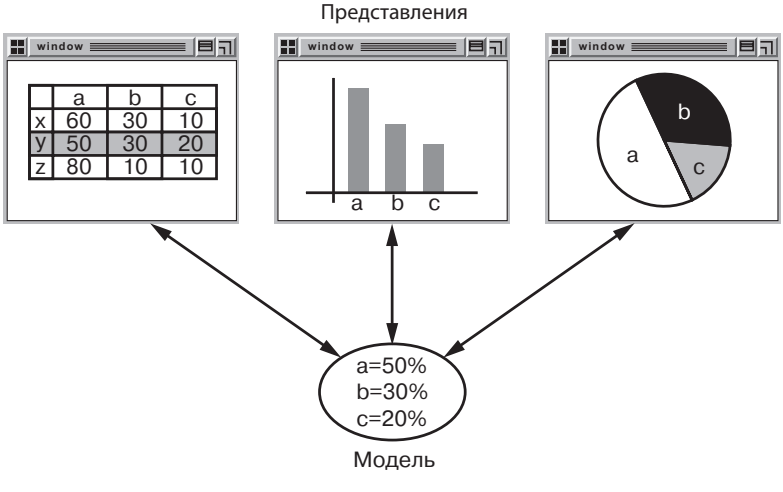
1.2. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В СХЕМЕ MVC В ЯЗЫКЕ SMALLTALK

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.

Builder (строитель) (124)

Отделяет конструирование сложного объекта от его представления, чтобы один процесс конструирования мог использоваться для создания различных представлений.

Chain of Responsibility (цепочка обязанностей) (263)

Можно избежать формирования жесткой связи между отправителем запроса и его получателем, для чего возможность обработки запроса предоставляется нескольким объектам. Объекты-получатели объединяются в цепочку, и запрос передается по цепочке, пока не будет обработан каким-либо объектом.

Command (команда) (275)

Инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов по типу запроса, ставить запросы в очередь, протоколировать их и поддерживать отмену выполнения операций.

Composite (компоновщик) (196)

Группирует объекты в древовидные структуры для представления иерархий типа «часть — целое». Позволяет клиентам работать с единичными объектами так же, как с группами объектов.

Decorator (декоратор) (209)

Динамически наделяет объект новыми обязанностями. Декораторы применяются для расширения существующей функциональности и являются гибкой альтернативой порождению подклассов.

Facade (фасад) (221)

Предоставляет унифицированный интерфейс к набору интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.

Factory Method (фабричный метод) (135)

Определяет интерфейс для создания объектов, позволяя подклассам решить, экземпляр какого класса следует создать. Позволяет классу передать ответственность за создание экземпляра в подклассы.

Flyweight (приспособленец) (231)

Применяет механизм совместного использования для эффективной поддержки большого числа мелких объектов.

Interpreter (интерпретатор) (287)

Для заданного языка определяет представление его грамматики вместе с интерпретатором, который использует представление для интерпретации предложений языка.

Iterator (итератор) (302)

Дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

Mediator (посредник) (319)

Определяет объект, в котором инкапсулирована информация о взаимодействии объектов из некоторого множества. Способствует ослаблению связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

Memento (хранитель) (330)

Позволяет без нарушения инкапсуляции получать и сохранять во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить в точно таком же состоянии.

Observer (наблюдатель) (339)

Определяет между объектами зависимость типа «один-ко-многим», так что при изменении состояния одного объекта все зависящие от него получают уведомление и автоматически обновляются.

Prototype (прототип) (146)

Описывает виды создаваемых объектов с помощью прототипа и создает новые объекты путем его копирования.

Proxy (заместитель) (246)

Подменяет другой объект для контроля доступа к нему.

Singleton (одиночка) (157)

Гарантирует, что некоторый класс может существовать только в одном экземпляре, и предоставляет глобальную точку доступа к нему.

State (состояние) (352)

Позволяет объекту изменять свое поведение при модификации внутреннего состояния. При этом все выглядит так, словно поменялся класс объекта.

Strategy (стратегия) (362)

Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Позволяет менять алгоритм независимо от клиента, который им пользуется.

Template Method (шаблонный метод) (373)

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять отдельные шаги алгоритма, не меняя его общей структуры.

Visitor (посетитель) (379)

Представляет операцию, которую надо выполнить над элементами объектной структуры. Позволяет определить новую операцию без изменения классов элементов, к которым он применяется.

1.5. ОРГАНИЗАЦИЯ КАТАЛОГА

Паттерны проектирования различаются степенью детализации и уровнем абстракции. Паттернов проектирования довольно много, поэтому их нужно как-то организовать. В данном разделе описывается классификация, позволяющая ссылаться на семейства взаимосвязанных паттернов. Она поможет быстрее освоить паттерны, описанные в каталоге, а также укажет направление поиска новых.

Мы будем классифицировать паттерны по двум критериям (табл. 1.1). Первый — *цель* — отражает назначение паттерна. Паттерны делятся на порождающие, структурные и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов

и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют.

Таблица 1.1. Пространство паттернов проектирования

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод (135)	Адаптер (171)	Интерпретатор (287) Шаблонный метод (373)
Объект	Абстрактная фабрика (113) Одиночка (157) Прототип (146) Строитель (124)	Адаптер (171) Декоратор (209) Заместитель (246) Компоновщик (196) Мост (184) Приспособленец (231) Фасад (221)	Итератор (302) Команда (275) Наблюдатель (339) Посетитель (379) Посредник (319) Состояние (352) Стратегия (362) Хранитель (330) Цепочка обязанностей (263)

Второй критерий — *уровень* — сообщает, к чему обычно применяется паттерн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции. Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения и потому более динамичны. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что концентрируются лишь на отношениях между классами. Обратите внимание: большинство паттернов действует на уровне объектов.

Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны

объектов описывают, как объекты, принадлежащие некоторой группе, совместными усилиями выполняют задачу, которая ни одному отдельному объекту не под силу.

Существуют и другие способы классификации паттернов. Некоторые паттерны часто используются вместе. Например, компоновщик применяется с итератором или посетителем. Некоторыми паттернами предлагаются альтернативные решения. Так, прототип нередко можно использовать вместо абстрактной фабрики. Применение части паттернов приводит к схожему дизайну, хотя изначально их назначение различно. Например, структурные диаграммы компоновщика и декоратора похожи.

Классифицировать паттерны можно и по их ссылкам (см. разделы «Родственные паттерны»). На рис. 1.1 такие отношения изображены графически.

Ясно, что организовать паттерны проектирования допустимо многими способами. Оценивая паттерны с разных точек зрения, вы глубже поймете, как они функционируют, как их сравнивать и когда применять тот или другой паттерн.

1.6. КАК РЕШАТЬ ЗАДАЧИ ПРОЕКТИРОВАНИЯ С ПОМОЩЬЮ ПАТТЕРНОВ

Паттерны проектирования позволяют решать многие повседневные задачи, с которыми сталкиваются проектировщики объектно-ориентированных приложений. Поясним эту мысль примерами.

ПОИСК ПОДХОДЯЩИХ ОБЪЕКТОВ

Объектно-ориентированные программы состоят из объектов. *Объект* сочетает данные и процедуры для их обработки. Такие процедуры обычно называют *методами* или *операциями*. Объект выполняет операцию, когда получает *запрос* (или *сообщение*) от клиента.

Отправка запроса — это *единственный* способ заставить объект выполнить операцию. А выполнение операции — *единственный* способ изменить внутреннее состояние объекта. Из-за этих двух ограничений говорят, что внутреннее состояние объекта инкапсулировано: к нему нельзя обратиться напрямую, а его представление невидимо за пределами объекта.

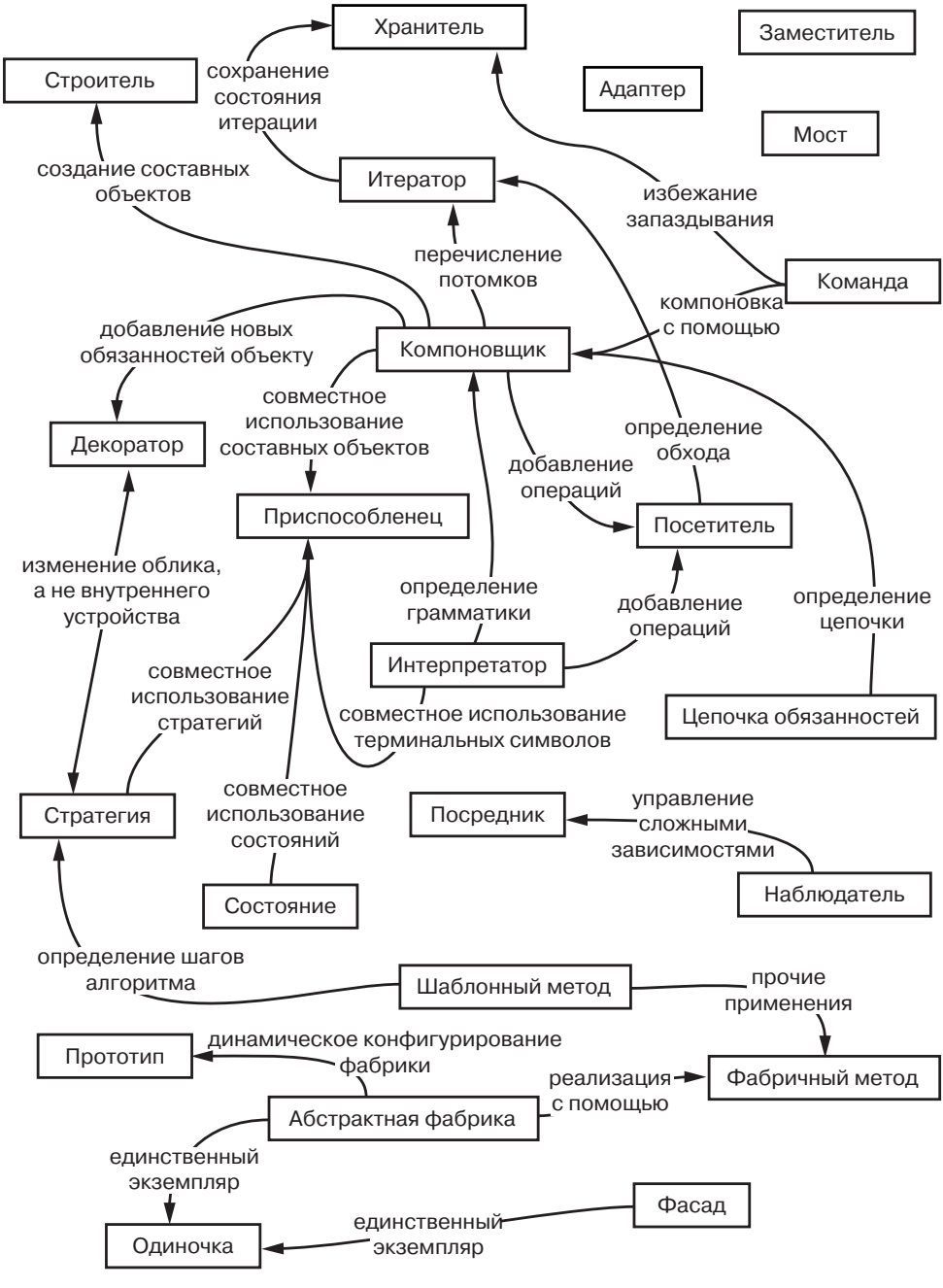


Рис. 1.1. Отношения между паттернами проектирования

Самая трудная задача в объектно-ориентированном проектировании — разложить систему на объекты. При решении приходится учитывать множество факторов: инкапсуляцию, глубину детализации, наличие зависимостей, гибкость, производительность, возможную эволюцию, повторное использование и т. д. и т. п. Все это влияет на декомпозицию, причем часто противоречивым образом.

Методологии объектно-ориентированного проектирования отражают разные подходы. Можно сформулировать задачу письменно, выделить из получившейся фразы существительные и глаголы, после чего создать соответствующие классы и операции. Другой путь — сосредоточиться на отношениях и разделении обязанностей в системе. Можно построить модель реального мира или перенести выявленные при анализе объекты в свой дизайн. Разработчики никогда не придут к единому мнению относительно того, какой подход самый лучший.

Многие объекты возникают в проекте из построенной в ходе анализа модели. Однако нередко появляются и классы, не имеющие аналогов в реальном мире. Это могут быть классы как низкого уровня, например массивы, так и высокого. Паттерн компоновщик (196) вводит такую абстракцию для единообразной трактовки объектов, у которой нет физического аналога. Если придерживаться строгого моделирования и ориентироваться только на реальный мир, то получится система, отражающая сегодняшние потребности, но, возможно, не учитывающая будущего развития. Абстракции, возникающие в ходе проектирования, — ключ к гибкому дизайну.

Паттерны проектирования помогают выявить не вполне очевидные абстракции и объекты, которые могут их использовать. Например, объектов, представляющих процесс или алгоритм, в действительности нет, но они являются неотъемлемыми составляющими гибкого дизайна. Паттерн стратегия (362) описывает способ реализации взаимозаменяемых семейств алгоритмов. Паттерн состояние (352) представляет состояние некоторой сущности в виде объекта. Эти объекты редко возникают во время анализа и даже на ранних стадиях проектирования. Они появляются позднее, при попытках сделать дизайн более гибким и пригодным для повторного использования.

ОПРЕДЕЛЕНИЕ СТЕПЕНИ ДЕТАЛИЗАЦИИ ОБЪЕКТА

Размеры и число объектов могут изменяться в широком диапазоне. С помощью объектов можно представить все, от физических устройств до программ. Как же решить, что должен представлять объект?

Паттерны проектирования помогут решить и эту проблему. Паттерн фасад (221) показывает, как представить в виде объекта целые подсистемы, а паттерн приспособленец (231) — как поддерживать большое число объектов при высокой степени детализации. Другие паттерны указывают путь к разложению объекта на меньшие подобъекты. Абстрактная фабрика (113) и строитель (124) описывают объекты, единственной целью которых является создание других объектов, а посетитель (379) и команда (275) — объекты, отвечающие за реализацию запроса к другому объекту или группе.

ОПРЕДЕЛЕНИЕ ИНТЕРФЕЙСОВ ОБЪЕКТА

Для любой операции, объявляемой объектом, должны быть заданы: имя операции, объекты, передаваемые в качестве параметров, и значение, возвращаемое операцией. Эту триаду называют *сигнатурой* операции. Множество сигнатур всех определенных для объекта операций называется *интерфейсом* этого объекта. Интерфейс описывает все множество запросов, которые можно отправить объекту. Любой запрос, сигнатура которого входит в интерфейс объекта, может быть ему отправлен.

Тип представляет собой имя, используемое для обозначения конкретного интерфейса. Говорят, что объект имеет тип `Window`, если он готов принимать запросы на выполнение любых операций, определенных в интерфейсе с именем `Window`. У одного объекта может быть много типов. Напротив, сильно отличающиеся объекты могут разделять общий тип. Одна часть интерфейса объекта может характеризоваться одним типом, а другие части — другими типами. Два объекта одного типа могут разделять только часть своих интерфейсов. Интерфейсы могут содержать другие интерфейсы в качестве подмножеств. Мы говорим, что один тип является *подтипом* другого, если интерфейс первого содержит интерфейс второго. В этом случае второй тип называется *супертипом* для первого. Часто говорят также, что подтип *наследует* интерфейс своего супертипа.

В объектно-ориентированных системах интерфейсы играют фундаментальную роль. Все взаимодействие с объектами осуществляется через их интерфейсы. Никакого способа получить информацию об объекте или заставить его что-то сделать в обход интерфейса не существует. Интерфейс объекта ничего не говорит о его реализации; разные объекты вправе реализовывать сходные запросы совершенно по-разному. Это означает, что два объекта с различными реализациями могут иметь одинаковые интерфейсы.

Когда объекту посылается запрос, то операция, которую он выполнит, зависит как от запроса, так и от объекта-адресата. Разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на такие запросы разные операции. Ассоциирование запроса с объектом и одной из его операций во время выполнения называется *динамическим связыванием*.

Динамическое связывание означает, что отправка некоторого запроса не определяет никакой конкретной реализации до момента выполнения. Следовательно, возможно написать программу, рассчитанную на объект с конкретным интерфейсом, точно зная, что любой объект с подходящим интерфейсом сможет принять этот запрос. Более того, динамическое связывание позволяет во время выполнения подставить вместо одного объекта другой, если он имеет идентичный интерфейс. Такая взаимозаменяемость называется *полиморфизмом* и является важнейшей особенностью объектно-ориентированных систем. Она позволяет клиенту ограничиваться минимальными предположениями об объектах — а именно поддержкой этими объектами определенного интерфейса. Полиморфизм упрощает определение клиентов, позволяет отделить объекты друг от друга и дает объектам возможность изменять отношения между ними во время выполнения.

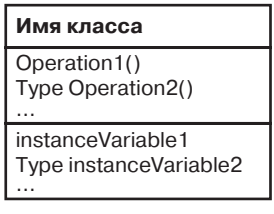
Паттерны проектирования позволяют определять интерфейсы посредством задания их основных элементов и того, какие данные можно передавать через интерфейс. Паттерн может также сообщить, что *не должно* включаться в интерфейс. Хорошим примером в этом отношении является хранитель (330). Он описывает, как инкапсулировать и сохранить внутреннее состояние объекта таким образом, чтобы в будущем его можно было восстановить точно в таком же состоянии. Объекты, удовлетворяющие требованиям паттерна хранитель, должны определить два интерфейса: один ограниченный, который позволяет клиентам держать у себя и копировать хранителей, а другой привилегированный, которым может пользоваться только исходный объект для сохранения и извлечения информации о своем состоянии в хранителе.

Паттерны проектирования также определяют отношения между интерфейсами. В частности, нередко они требуют, чтобы некоторые классы имели схожие интерфейсы, а иногда налагают ограничения на интерфейсы классов. Так, декоратор (209) и заместитель (246) требуют, чтобы интерфейсы объектов этих паттернов были идентичны интерфейсам декорируемых и замещаемых объектов соответственно. Интерфейс объекта, использующего паттерн посетитель (379), должен отражать все классы объектов, с которыми он будет работать.

ОПРЕДЕЛЕНИЕ РЕАЛИЗАЦИИ ОБЪЕКТОВ

До сих пор мы почти ничего не сказали о том, как же в действительности определяется объект. Реализация объекта определяется его *классом*. Класс определяет внутренние данные объекта и его представление, а также операции, которые объект может выполнять.

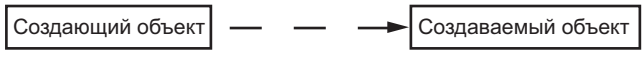
В нашей нотации, основанной на OMT (см. приложение Б), класс изображается прямоугольником, внутри которого жирным шрифтом написано имя класса. Ниже обычным шрифтом перечислены операции. Все данные, определенные для класса, следуют после операций. Имя класса, операции и данные разделяются горизонтальными линиями:



Типы возвращаемого значения и переменных экземпляра необязательны, поскольку мы не ограничиваем себя языками программирования с сильной типизацией.

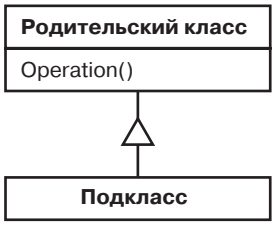
Объекты создаются посредством *создания экземпляров* класса. Говорят, что объект является экземпляром класса. В процессе создания экземпляров выделяется память для внутренних данных объекта (*переменных экземпляра*), а с этими данными связываются операции. Создавая экземпляры одного класса, можно создать много сходных объектов.

Пунктирная линия со стрелкой обозначает класс, который создает объекты другого класса. Стрелка направлена в сторону класса создаваемых объектов.



Новые классы могут определяться в контексте существующих с использованием *наследования классов*. Если *подкласс* наследует *родительскому классу*, то он включает определения всех данных и операций, определенных в родительском классе. Объекты, являющиеся экземплярами подкласса, будут содержать все данные, определенные как в самом подклассе, так и во всех

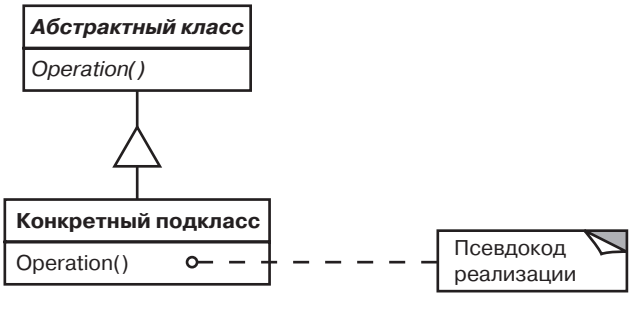
его родительских классах. Такой объект сможет выполнять все операции, определенные в этом подклассе и его предках. Отношение *«является подклассом»* обозначается вертикальной линией с треугольником.



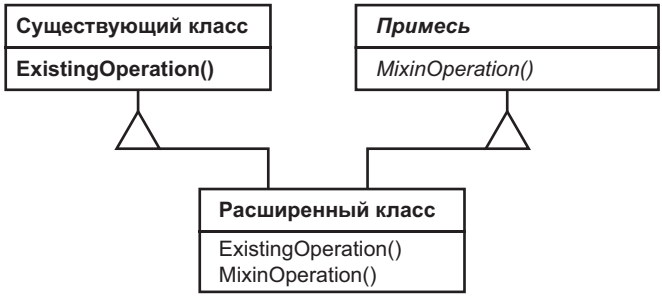
Класс называется абстрактным, если его единственное назначение — определить общий интерфейс для всех своих подклассов. Абстрактный класс делегирует реализацию всех или части своих операций подклассам, поэтому у него не может быть экземпляров. Операции, объявленные, но не реализованные в абстрактном классе, называются *абстрактными*. Класс, не являющийся абстрактным, называется *конкретным*.

Подклассы могут уточнять или переопределять поведение своих предков. Точнее, класс может *заместить* операцию, определенную в родительском классе. Замещение дает подклассам возможность обрабатывать запросы, адресованные родительским классам. Наследование позволяет определять новые классы, просто расширяя возможности старых. Таким образом можно без труда определять семейства объектов, обладающих сходной функциональностью.

Имена абстрактных классов записываются курсивом, чтобы отличать их от конкретных. Курсив используется также для обозначения абстрактных операций. На диаграмме может изображаться псевдокод, описывающий реализацию операции; в таком случае код представлен в прямоугольнике с загнутым уголком, соединенном пунктирной линией с операцией, которую он реализует.



Примесью (mixin class) называется класс, назначение которого — предоставить дополнительный интерфейс или функциональность другим классам. Он отчасти похож на абстрактные классы в том смысле, что не предполагает непосредственного создания экземпляров. Для работы с примесями необходимо множественное наследование:



НАСЛЕДОВАНИЕ КЛАССА И НАСЛЕДОВАНИЕ ИНТЕРФЕЙСА

Важно понимать различие между *классом* объекта и его *типом*.

Класс объекта определяет реализацию объекта, то есть внутреннее состояние и реализацию операций объекта. Напротив, тип относится только к интерфейсу объекта — множеству запросов, на которые объект способен ответить. У объекта может быть много типов, и объекты разных классов могут иметь один и тот же тип.

Разумеется, между классом и типом существует тесная связь. Поскольку класс определяет операции, которые может выполнять объект, то он также определяет и его тип. Когда мы говорим «объект является экземпляром класса», то подразумеваем, что он поддерживает интерфейс, определяемый этим классом.

В языках вроде C++ и Eiffel классы определяют как тип объекта, так и его реализацию. В программах на языке Smalltalk типы переменных не объявляются, поэтому компилятор не проверяет, что тип объекта, присваиваемого переменной, является подтипом типа переменной. При отправке сообщения необходимо проверять, что класс получателя реализует реакцию на сообщение, но проверка того, что получатель является экземпляром определенного класса, не нужна.

Важно также понимать различие между наследованием класса и наследованием интерфейса (или порождением подтипов). В случае наследования

класса реализация объекта определяется в терминах реализации другого объекта. Проще говоря, это механизм разделения кода и представления. Напротив, наследование интерфейса (порождение подтипов) описывает, когда один объект можно использовать вместо другого.

Эти две концепции легко спутать, поскольку во многих языках явное различие отсутствует. В таких языках, как C++ и Eiffel, под наследованием понимается одновременно наследование интерфейса и реализации. Стандартный способ реализации наследования интерфейса в C++ — это открытое наследование классу, в котором есть (чисто) виртуальные функции. Чистое наследование интерфейса можно смоделировать в C++ посредством открытого наследования чисто абстрактному классу. Чистая реализация или наследование классов может моделироваться посредством закрытого наследования. В Smalltalk под наследованием понимается только наследование реализации. Переменной можно присвоить экземпляры любого класса при условии, что они поддерживают операции, выполняемые над значением этой переменной.

Хотя в большинстве языков программирования различие между наследованием интерфейса и реализации не поддерживается, на практике оно существует. Программисты на Smalltalk обычно считают, что подклассы — это подтипы (хотя имеются и хорошо известные исключения [Coo92]). Программисты на C++ манипулируют объектами через типы, определяемые абстрактными классами.

Многие паттерны проектирования зависят от этого различия. Например, объекты, построенные в соответствии с паттерном цепочка обязанностей (263), должны иметь общий тип, но обычно они не используют общую реализацию. В паттерне компоновщик (196) отдельный объект (компонент) определяет общий интерфейс, но реализацию часто определяет составной объект (композиция). Паттерны команда (275), наблюдатель (339), состояние (352) и стратегия (362) часто реализуются абстрактными классами, которые представляют чистые интерфейсы.

Программирование в соответствии с интерфейсом, а не с реализацией

Наследование классов — это не что иное, как механизм расширения функциональности приложения путем повторного использования функциональности родительских классов. Оно позволяет быстро определить новый вид объектов через уже имеющийся. Новую реализацию можно получить почти без всякого труда посредством наследования большей части необходимого кода из существующих классов.

Впрочем, повторное использование реализации — лишь полдела. Не менее важно, что наследование позволяет определять семейства объектов с *идентичными* интерфейсами (обычно за счет наследования от абстрактных классов). Почему? Потому что от этого зависит полиморфизм.

Если пользоваться наследованием осторожно (некоторые сказали бы *правильно*), то все классы, производные от некоторого абстрактного класса, будут обладать его интерфейсом. Отсюда следует, что подкласс добавляет новые или замещает старые операции и не скрывает операции, определенные в родительском классе. *Все* подклассы могут отвечать на запросы, соответствующие интерфейсу абстрактного класса, поэтому они являются подтипами этого абстрактного класса.

У манипулирования объектами строго через интерфейс абстрактного класса есть два преимущества:

- клиенту не нужно располагать информацией о конкретных типах объектов, которыми он пользуется, при условии что все они имеют ожидаемый клиентом интерфейс;
- клиенту необязательно «знать» о классах, с помощью которых реализованы объекты. Клиенту известно только об абстрактном классе (или классах), определяющих интерфейс.

Данные преимущества настолько существенно уменьшают число зависимостей между подсистемами, что можно даже сформулировать принцип объектно-ориентированного проектирования для повторного использования:

программируйте в соответствии с интерфейсом, а не с реализацией.

Не объявляйте переменные как экземпляры конкретных классов. Вместо этого придерживайтесь интерфейса, определенного абстрактным классом. Этот принцип проходит через все паттерны, описанные в книге.

Конечно, где-то в системе вам придется создавать экземпляры конкретных классов, то есть определить конкретную реализацию. Как раз это и позволяют сделать порождающие паттерны: абстрактная фабрика (113), строитель (124), фабричный метод (135), прототип (146) и одиночка (157). Абстрагируя процесс создания объекта, эти паттерны предоставляют вам разные способы прозрачного связывания интерфейса с его реализацией в момент создания экземпляра. Использование порождающих паттернов гарантирует, что система написана в категориях интерфейсов, а не реализации.

МЕХАНИЗМЫ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ

Большинству проектировщиков известны концепции объектов, интерфейсов, классов и наследования. Трудность в том, чтобы применить эти знания для построения гибких, повторно используемых программ. С помощью паттернов проектирования вам будет проще это сделать.

Наследование и композиция

Два наиболее распространенных приема повторного использования функциональности в объектно-ориентированных системах — это наследование класса и *композиция объектов*. Как мы уже объясняли, наследование класса позволяет определить реализацию одного класса через другой. Повторное использование за счет порождения подкласса называют еще *повторным использованием по принципу прозрачного ящика* (white box reuse). Такой термин подчеркивает, что внутреннее устройство родительских классов часто видимо подклассам.

Композиция объектов — альтернатива наследованию класса. В этом случае новая, более сложная функциональность получается путем объединения или *композиции* объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ называют *повторным использованием по принципу черного ящика* (blackbox reuse), поскольку детали внутреннего устройства объектов остаются скрытыми.

И у наследования, и у композиции есть достоинства и недостатки. Наследование класса определяется статически на этапе компиляции, его проще использовать, поскольку оно напрямую поддержано языком программирования. В случае наследования классов упрощается также задача модификации существующей реализации. Если подкласс замещает лишь некоторые операции, то могут оказаться затронутыми и остальные унаследованные операции (при условии что они вызывают замещенные).

Впрочем, у наследования класса есть и минусы. Во-первых, унаследованную от родителя реализацию не удастся изменить во время выполнения программы, поскольку наследование определяется на этапе компиляции. Во-вторых (и это более серьезно), родительский класс нередко хотя бы частично определяет физическое представление своих подклассов. Поскольку подклассу доступны детали реализации родительского класса, то часто говорят, что наследование нарушает инкапсуляцию [Sny86]. Реализации подкласса и родительского класса связываются настолько тесно, что любые изменения последней требуют изменять и реализацию подкласса.

Зависимость от реализации может повлечь за собой проблемы при попытке повторного использования подкласса. Если хотя бы один аспект унаследованной реализации непригоден для новой предметной области, то приходится переписывать родительский класс или заменять его чем-то более подходящим. Такая зависимость ограничивает гибкость и возможности повторного использования. С проблемой можно справиться, если наследовать только абстрактным классам, поскольку в них обычно совсем нет реализации или она минимальна.

Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композиция требует, чтобы объекты соблюдали интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик: поскольку доступ к объектам осуществляется только через их интерфейсы, инкапсуляция не нарушается. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку реализация объекта пишется прежде всего в категориях его интерфейсов, то зависимость от реализации резко снижается.

Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов перед наследованием классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), а поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе.

Это подводит нас ко второму правилу объектно-ориентированного проектирования:

предпочитайте композицию наследованию класса.

В идеале для достижения повторного использования вообще не следовало бы создавать новые компоненты. Было бы лучше, если бы всю необходимую функциональность можно было получить простым объединением уже существующих компонентов. На практике, однако, так получается редко, поскольку набор имеющихся компонентов все же недостаточно широк. Повторное использование за счет наследования упрощает создание новых

компонентов, которые можно было бы применять со старыми. Поэтому наследование и композиция часто используются вместе.

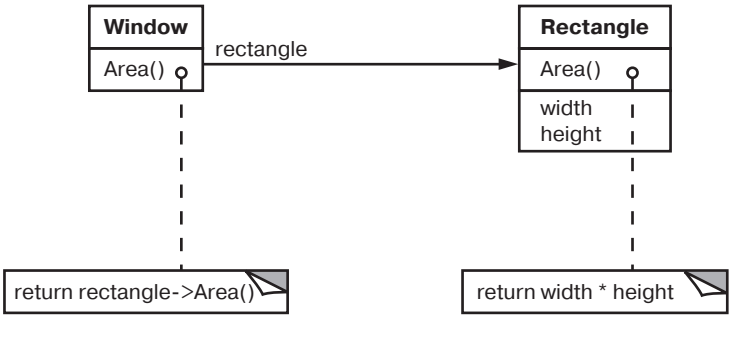
Тем не менее наш опыт показывает, что проектировщики злоупотребляют наследованием. Нередко дизайн мог бы стать лучше (и проще), если бы автор больше полагался на композицию объектов. Композиция будет снова и снова встречаться вам в паттернах этой книги.

Делегирование

С помощью делегирования композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование [Lie86, JZ91]. При делегировании в процессе обработки запроса задействованы *два* объекта: получатель поручает выполнение операций другому объекту — *уполномоченному* (*делегату*). Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную класса (в C++) или переменную `self` (в Smalltalk). Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту, дабы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса.

Например, вместо того чтобы делать класс `Window` (окно) подклассом класса `Rectangle` (прямоугольник) — ведь окно является прямоугольником, — мы можем воспользоваться внутри `Window` поведением класса `Rectangle`, поместив в класс `Window` переменную экземпляра типа `Rectangle` и *делегируя* ей операции, специфические для прямоугольников. Другими словами, окно не *является* прямоугольником, а *содержит* его. Теперь класс `Window` может явно перенаправлять запросы своей переменной `Rectangle`, а не наследовать ее операции.

На схеме ниже изображен класс `Window`, который делегирует операцию `Area()` переменной экземпляра `Rectangle`.



Сплошная линия со стрелкой обозначает, что класс содержит ссылку на экземпляр другого класса. Эта ссылка может иметь необязательное имя, в данном случае `rectangle`.

Главное достоинство делегирования в том, что оно упрощает композицию поведений во время выполнения. При этом способ комбинирования поведений можно изменять. Например, внутреннюю область окна можно преобразовать в круг во время выполнения, просто подставив вместо экземпляра класса `Rectangle` экземпляр класса `Circle` (предполагается, конечно, что оба эти класса имеют одинаковый тип).

У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Динамическую программу с высокой степенью параметризации труднее понять, нежели статическую. Также присутствует и некоторая потеря машинной эффективности, но в долгосрочной перспективе неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим вариантом только тогда, когда оно позволяет достичь упрощения, а не усложнения дизайна. Нелегко сформулировать правила, которые бы однозначно определяли, когда следует пользоваться делегированием, поскольку эффективность его зависит от контекста и вашего личного опыта. Лучше всего делегирование работает при использовании в составе привычных идиом, то есть в стандартных паттернах.

Делегирование используется в нескольких паттернах проектирования: состояние (352), стратегия (362), посетитель (379). В первом объект делегирует запрос объекту, представляющему его текущее состояние. В паттерне стратегия обработка запроса делегируется объекту, который представляет стратегию его исполнения. У объекта может быть только одно состояние, но много стратегий для исполнения различных запросов. Назначение обоих паттернов — изменить поведение объекта за счет замены объектов, которым делегируются запросы. В паттерне посетитель операция, которая должна быть выполнена над каждым элементом составного объекта, всегда делегируется посетителю.

В других паттернах делегирование используется не так интенсивно. Паттерн посредник (319) вводит объект, осуществляющий посредничество при взаимодействии других объектов. Иногда объект-посредник реализует операции, переадресуя их другим объектам; в других случаях он передает ссылку на самого себя, используя тем самым делегирование как таковое. Паттерн цепочка обязанностей (263) обрабатывает запросы, перенаправляя их от одного объекта другому по цепочке. Иногда вместе с запросом передается

ссылка на исходный объект, получивший запрос, и в этом случае мы снова сталкиваемся с делегированием. Паттерн мост (184) отделяет абстракцию от ее реализации. Если между абстракцией и конкретной реализацией имеется существенное сходство, то абстракция может просто делегировать операции своей реализации.

Делегирование — особый случай композиции. Оно показывает, что наследование как механизм повторного использования всегда можно заменить композицией.

Наследование и параметризованные типы

Еще один (хотя и не в точности объектно-ориентированный) метод повторного использования имеющейся функциональности — это применение *параметризованных* типов, известных также как обобщенные типы (Ada, Eiffel) или шаблоны (C++). Данная техника позволяет определить тип, не задавая типы, которые он использует. Отсутствующие типы передаются в *параметрах* в точке использования. Например, класс `List` (список) можно параметризовать типом помещаемых в список элементов. Чтобы объявить список целых чисел, вы передаете тип `integer` в качестве параметра параметризованному типу `List`. Если же надо объявить список строк, то в качестве параметра передается тип `String`. Для каждого типа элементов компилятор языка создаст отдельный вариант шаблона класса `List`.

Параметризованные типы предоставляют третий (после наследования класса и композиции объектов) способ комбинировать поведение в объектно-ориентированных системах. Многие задачи можно решить с помощью любого из этих трех методов. Чтобы параметризовать процедуру сортировки операцией сравнения элементов, сравнение можно было бы сделать:

- операцией, реализуемой подклассами (применение паттерна шаблонный метод (373));
- функцией объекта, передаваемого процедуре сортировки (стратегия (362));
- аргументом шаблона в C++ или обобщенного типа в Ada, который задает имя функции, вызываемой для сравнения элементов.

Но между этими тремя подходами есть важные различия. Композиция объектов позволяет изменять поведение во время выполнения, но для этого требуются косвенные вызовы, что снижает эффективность. Наследование разрешает предоставить реализацию по умолчанию, которую можно замещать в подклассах. Параметризованные типы позволяют изменять типы,

используемые классом. Но ни наследование, ни параметризованные типы не могут изменяться во время выполнения. Выбор того или иного подхода зависит от проекта и ограничений реализации.

Ни в одном из паттернов, описанных в этой книге, параметризованные типы не используются, хотя изредка мы прибегаем к ним для реализации паттернов в C++. В языке вроде Smalltalk, где нет проверки типов во время компиляции, параметризованные типы не нужны вовсе.

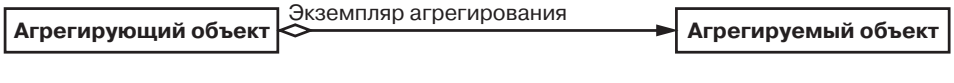
СРАВНЕНИЕ СТРУКТУР ВРЕМЕНИ ВЫПОЛНЕНИЯ И ВРЕМЕНИ КОМПИЛЯЦИИ

Структура объектно-ориентированной программы на этапе выполнения часто имеет мало общего со структурой ее исходного кода. Последняя фиксируется на этапе компиляции; код состоит из классов, отношения наследования между которыми неизменны. На этапе же выполнения структура программы — быстро изменяющаяся сеть из взаимодействующих объектов. Две эти структуры почти независимы. Пытаться понять одну по другой — все равно что пытаться понять динамику живых экосистем по статической таксономии растений и животных, или наоборот.

Рассмотрим различие между *агрегированием* и *осведомленностью* (acquaintance) объектов и его проявления на этапах компиляции и выполнения. Агрегирование подразумевает, что один объект владеет другим или несет за него ответственность. В общем случае мы говорим, что объект *содержит* другой объект или *является его частью*. Агрегирование означает, что агрегат и его составляющие имеют одинаковое время жизни.

Говоря же об осведомленности, мы имеем в виду, что объекту *известно* о другом объекте. Иногда осведомленность называют ассоциацией или отношением «использует». Осведомленные объекты могут запрашивать друг у друга операции, но они не несут никакой ответственности друг за друга. Осведомленность — это более слабое отношение, чем агрегирование; оно предполагает гораздо менее тесную связь между объектами.

На наших схемах осведомленность будет обозначаться сплошной линией со стрелкой. Линия со стрелкой и ромбиком в начале обозначает агрегирование.



Агрегирование и осведомленность легко спутать, поскольку они часто реализуются одинаково. В языке Smalltalk все переменные являются ссылками на объекты, здесь нет различия между агрегированием и осведомленностью. В C++ агрегирование можно реализовать путем определения переменных класса, которые являются реальными экземплярами, но чаще они определяются в виде указателей или ссылок. Осведомленность также реализуется с помощью указателей и ссылок.

Различие между осведомленностью и агрегированием в конечном итоге определяется, скорее, предполагаемым использованием, а не языковыми механизмами. В структуре, существующей на этапе компиляции, увидеть различие нелегко, но тем не менее оно существенно. Обычно отношений агрегирования меньше, чем отношений осведомленности, и они более постоянны. Напротив, отношения осведомленности возникают и исчезают чаще и иногда длятся лишь во время исполнения некоторой операции. Кроме того, отношения осведомленности более динамичны, что затрудняет их выявление в исходном тексте программы.

Коль скоро несоответствие между структурой программы на этапах компиляции и выполнения столь велико, ясно, что изучение исходного кода может сказать о работе системы совсем немного. Структура системы на стадии выполнения должно определяться проектировщиком, а не языком. Соотношения между объектами и их типами нужно проектировать очень аккуратно, поскольку именно от них зависит, насколько удачной или неудачной окажется структура во время выполнения.

Многие паттерны проектирования (особенно уровня объектов) явно подчеркивают различие между структурами на этапах компиляции и выполнения. Паттерны компоновщик (196) и декоратор (209) особенно полезны для построения сложных структур времени выполнения. В работе паттерна наблюдатель (339) задействованы структуры времени выполнения, которые часто трудно понять, не зная паттерна. Паттерн цепочка обязанностей (263) также приводит к таким схемам взаимодействия, в которых наследование неочевидно. В общем можно утверждать, что без понимания специфики паттернов разобраться в структурах времени выполнения невозможно.

ПРОЕКТИРОВАНИЕ С УЧЕТОМ БУДУЩИХ ИЗМЕНЕНИЙ

Системы должны проектироваться с учетом их дальнейшего развития. Для проектирования системы, устойчивой к таким изменениям, следует предположить, как она будет изменяться на протяжении отведенного ей времени жизни.

Если при проектировании системы не принималась во внимание возможность изменений, то есть вероятность, что в будущем ее придется полностью перепроектировать. Это может повлечь за собой переопределение и новую реализацию классов, модификацию клиентов и повторный цикл тестирования. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими.

Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида.

Вот некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать:

- *при создании объекта явно указывается класс.* Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно.

  Паттерны проектирования: абстрактная фабрика (113), фабричный метод (135), прототип (146);

- *зависимость от конкретных операций.* Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения.

  Паттерны проектирования: цепочка обязанностей (263), команда (275);

- *зависимость от аппаратной и программной платформ.* Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Возможно, даже на «родной» платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

  Паттерны проектирования: абстрактная фабрика (113), мост (184);

- *зависимость от представления или реализации объекта.* Если клиент располагает информацией о том, как объект представлен, хранится или реализован, то, возможно, при изменении объекта придется изменять

и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскадных изменений.

Паттерны проектирования: абстрактная фабрика (113), мост (184), хранитель (330), заместитель (246);

- *зависимость от алгоритмов.* Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, которые с большой вероятностью будут изменяться, следует изолировать.

  Паттерны проектирования: мост (184), итератор (302), стратегия (362), шаблонный метод (373), посетитель (379);

- *сильная связанность.* Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать.

  Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабосвязанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

  Паттерны проектирования: абстрактная фабрика (113), мост (184), цепочка обязанностей (263), команда (275), фасад (221), посредник (319), наблюдатель (339);

- *расширение функциональности за счет порождения подклассов.* Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т. д.). Для определения подкласса необходимо так же ясно представлять себе устройство родительского класса. Например, замещение одной операции может потребовать замещения и других. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к разрастанию количества классов, поскольку даже для реализации простого расширения приходится создавать новые подклассы.

Композиция объектов и делегирование — гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, интенсивное использование композиции объектов может усложнить понимание кода. С помощью многих паттернов проектирования удается построить такое решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими.

Паттерны проектирования: мост (184), цепочка обязанностей (263), компоновщик (196), декоратор (209), наблюдатель (339), стратегия (362);

- *неудобства при изменении классов*. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а он недоступен (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.

  Паттерны проектирования: адаптер (171), декоратор (209), посетитель (379).

Приведенные примеры демонстрируют ту гибкость, которой можно добиться, используя паттерны при проектировании приложений. Насколько эта гибкость необходима, зависит, конечно, от особенностей вашей программы. Давайте посмотрим, какую роль играют паттерны при разработке приложений, инструментальных библиотек и каркасов приложений.

Приложения

Если вы проектируете приложения — например, редактор документов или электронную таблицу, — то наивысший приоритет имеют *внутреннее* повторное использование, удобство сопровождения и расширяемость. Первое подразумевает, что вы не проектируете и не реализуете больше, чем необходимо. Повысить степень внутреннего повторного использования помогут паттерны, уменьшающие число зависимостей. Ослабление связанности увеличивает вероятность того, что некоторый класс объектов сможет взаимодействовать с другими. Например, устраняя зависимости от конкретных операций путем изолирования и инкапсуляции каждой операции, вы упрощаете задачу повторного использования любой операции в другом контексте. К тому же результату приводит устранение зависимостей от алгоритма и представления.

Паттерны проектирования также упрощают сопровождение приложения, если использовать их для ограничения платформенных зависимостей и разбиения системы на уровни. Они способствуют и наращиванию функциональности системы, показывая, как расширять иерархии классов и когда следует применять композицию объектов. Уменьшение степени связанности также увеличивает возможность развития системы. Расширение класса становится проще, если он не зависит от множества других.

Инструментальные библиотеки

Часто приложение включает классы из одной или нескольких библиотек заранее определенных классов. Такие библиотеки называются *инструментальными*. Инструментальная библиотека — это набор взаимосвязанных, повторно используемых классов, спроектированный с целью предоставления полезной функциональности общего назначения. Примеры инструментальной библиотеки — набор контейнерных классов для списков, ассоциативных массивов, стеков и т. д., библиотека потокового ввода/вывода в C++. Инструментальные библиотеки не определяют какой-то конкретный дизайн приложения, а просто предоставляют средства, упрощающие решение определенных задач в приложениях, позволяют разработчику не изобретать стандартную функциональность. Таким образом, в инструментальных библиотеках упор делается на *повторном использовании кода*. Это объектно-ориентированные эквиваленты библиотек подпрограмм.

Существует мнение, что проектировать инструментальные библиотеки сложнее, чем приложения, поскольку библиотеки должны использоваться во многих приложениях (иначе они бесполезны.) К тому же автор библиотеки не знает заранее, какие специфические требования будут предъявляться конкретными приложениями. Поэтому ему необходимо избегать любых предположений и зависимостей, способных ограничить гибкость библиотеки — а следовательно, сферу ее применения и эффективность.

Каркасы приложений

Каркас — это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ [Deu89, JF88]. Например, можно создать каркас для разработки графических редакторов в разных областях: рисовании, сочинении музыки или САПР [VL90, Joh92]. Другой каркас может специализироваться на создании компиляторов для разных языков программирования и целевых машин [JML92]. Третий упростит построение приложений для финансового моделирования [BE93]. Каркас можно адаптировать для конкретного приложения

путем порождения специализированных подклассов от входящих в него абстрактных классов.

Каркас диктует определенную архитектуру приложения. Он определяет общую структуру, ее разделение на классы и объекты, ключевые обязанности тех и других, методы взаимодействия объектов и классов и потоки управления. Данные параметры проектирования задаются каркасом, а проектировщики или разработчики приложений могут сконцентрироваться на специфике приложения. В каркасе отражены проектные решения, общие для данной предметной области. Акцент в каркасе делается на *повторном использовании дизайна*, а не кода, хотя обычно он включает и конкретные подклассы, которые можно применять непосредственно.

Повторное использование на данном уровне означает *инверсию контроля* между приложением и программным обеспечением, лежащим в его основе. При использовании инструментальной библиотеки (или, если хотите, обычной библиотеки подпрограмм) вы пишете основной код приложения и вызываете из него код, который планируете использовать повторно. При работе с каркасом вы, наоборот, повторно используете основной код и пишете код, который *он* вызывает. Вам приходится кодировать операции с предопределенными именами и параметрами вызова, но зато число принимаемых вами проектных решений сокращается.

В результате не только ускоряется построение приложений, но и все приложения имеют схожую структуру. Их проще сопровождать, и пользователям они представляются более знакомыми. С другой стороны, вы в какой-то мере жертвуете свободой творчества, поскольку многие проектные решения уже приняты за вас.

Если проектировать приложения нелегко, инструментальные библиотеки — еще сложнее, то проектирование каркасов — задача самая трудная. Проектировщик каркаса рассчитывает, что единая архитектура будет пригодна для всех приложений в данной предметной области. Любое независимое изменение дизайна каркаса приведет к утрате его преимуществ, поскольку основной «вклад» каркаса в приложение — это определяемая им архитектура. Поэтому каркас должен быть максимально гибким и расширяемым.

Поскольку приложения так сильно зависят от каркаса, они особенно чувствительны к изменениям его интерфейсов. По мере усложнения каркаса приложения должны эволюционировать вместе с ним. В результате существенно возрастает значение слабой связанности, в противном случае малейшее изменение каркаса приведет к целой волне модификаций.

Рассмотренные выше проблемы проектирования критичны именно для каркасов. Каркас, в котором они решены путем применения паттернов, может лучше обеспечить высокий уровень проектирования и повторного использования кода, чем тот, где паттерны не применялись. В каркасах, прошедших проверку временем, обычно задействовано несколько разных паттернов проектирования. Паттерны помогают адаптировать архитектуру каркаса ко многим приложениям без повторного проектирования.

Дополнительное преимущество проявляется при документировании каркаса с указанием тех паттернов, которые в нем использованы [BJ94]. Тот, кто знает паттерны, способен быстрее разобраться в тонкостях каркаса. Но даже не работающие с паттернами увидят их преимущества, поскольку паттерны помогают удобно структурировать документацию по каркасу. Повышение качества документирования важно для любых программных продуктов, но для каркасов этот аспект важен вдвойне. Для освоения работы с каркасами надо потратить немало усилий, и только после этого они начнут приносить реальную пользу. Паттерны могут существенно упростить задачу, явно выделяя ключевые элементы дизайна каркаса.

Поскольку между паттернами и каркасами много общего, часто возникает вопрос, в чем же различия между ними и есть ли они вообще. Так вот, существуют три основных различия:

- *паттерны проектирования более абстрактны, чем каркасы.* В код могут быть включены целые каркасы, но только отдельные *воплощения* паттернов. Каркасы можно писать на разных языках программирования и не только изучать, но и непосредственно исполнять и повторно использовать. В противоположность этому паттерны проектирования, описанные в данной книге, необходимо реализовывать всякий раз, когда в них возникает необходимость. Паттерны объясняют намерения проектировщика, сильные и слабые стороны, а также последствия выбранного дизайна;
- *как архитектурные элементы, паттерны проектирования мельче, чем каркасы.* Типичный каркас содержит несколько паттернов. Обратное утверждение неверно;
- *паттерны проектирования менее специализированы, чем каркасы.* Каркас всегда создается для конкретной предметной области. В принципе каркас графического редактора можно использовать для моделирования работы фабрики, но его никогда не спутаешь с каркасом, предназначенным специально для моделирования. Напротив, включенные в наш каталог паттерны могут использоваться в приложениях почти любого

вида. Хотя, безусловно, существуют и более специализированные паттерны (скажем, паттерны для распределенных систем или параллельного программирования), но даже они не диктуют выбор архитектуры приложения в той же мере, что и каркасы.

Каркасы встречаются все чаще, а их роль в разработке растет. Именно с их помощью объектно-ориентированные системы можно использовать повторно в максимальной степени. Крупные объектно-ориентированные приложения в конечном итоге строятся из каркасов, взаимодействующих друг с другом на разных уровнях. Дизайн и код приложения в значительной мере определяются теми каркасами, которые применялись при его создании.

1.7. КАК ВЫБИРАТЬ ПАТТЕРН ПРОЕКТИРОВАНИЯ

Если учесть, что каталог содержит более 20 паттернов, выбрать паттерн, лучше всего подходящий для решения конкретной задачи проектирования, будет непросто. Ниже представлены некоторые подходы к выбору подходящего паттерна:

- *подумайте, как паттерны решают проблемы проектирования.* В разделе 1.6 обсуждается то, как с помощью паттернов найти подходящие объекты, определить нужную степень их детализации, специфицировать их интерфейсы. Здесь же говорится и о некоторых иных подходах к решению задач с помощью паттернов;
- *пролистайте разделы каталога, описывающие назначение паттернов.* В разделе 1.4 (с. 24) перечислены назначения всех представленных паттернов. Ознакомьтесь с назначением каждого паттерна, когда будете искать тот, что в наибольшей степени относится к вашей проблеме. Чтобы сузить поиск, воспользуйтесь схемой в таблице 1.1 (с. 28);
- *изучите взаимосвязи паттернов.* На рис. 1.1 (с. 30) графически изображены соотношения между различными паттернами проектирования. Данная информация поможет вам найти нужный паттерн или группу паттернов;
- *проанализируйте паттерны со сходными целями.* Каталог (с. 108) состоит из трех частей: порождающие паттерны, структурные паттерны и паттерны поведения. Каждая часть начинается со вступительных замечаний о паттернах соответствующего вида и заканчивается разделом, где они сравниваются друг с другом. Эти разделы позволяют лучше по-

нять сходства и различия между паттернами, имеющими похожее назначение;

- *разберитесь в причинах, вызывающих перепроектирование.* Взгляните на перечень причин, приведенный выше, и проверьте, нет ли в нем вашей проблемы. Затем обратитесь к описаниям паттернов, помогающим устранить эту причину;
- *посмотрите, какие аспекты вашего дизайна могут измениться.* Такой подход противоположен анализу причин, вызвавших необходимость перепроектирования. Вместо того чтобы думать, что могло бы *заставить* изменить дизайн, подумайте о том, что бы вам хотелось *иметь возможность* изменять без перепроектирования. Акцент здесь делается на *инкапсуляции концепций, подверженных изменениям* — основной теме многих паттернов. В табл. 1.2 перечислены те аспекты дизайна, которые разные паттерны позволяют модифицировать независимо, чтобы вы могли изменять их без перепроектирования.

Таблица 1.2. Аспекты дизайна, которые могут изменяться при применении паттернов проектирования

Назначение	Паттерн проектирования	Переменные аспекты
Порождающие паттерны	Абстрактная фабрика (113)	Семейства порождаемых объектов
	Одиночка (157)	Единственный экземпляр класса
	Прототип (146)	Класс, на основе которого создается объект
	Строитель (124)	Способ создания составного объекта
	Фабричный метод (135)	Подкласс создаваемого объекта
Структурные паттерны	Адаптер (171)	Интерфейс к объекту
	Декоратор (209)	Обязанности объекта без порождения подкласса
	Заместитель (246)	Способ доступа к объекту, его местоположение
	Компоновщик (196)	Структура и состав объекта
	Мост (184)	Реализация объекта
	Приспособленец (231)	Затраты на хранение объектов
	Фасад (221)	Интерфейс к подсистеме

Таблица 1.2 (*окончание*)

Назначение	Паттерн проектирования	Переменные аспекты
Паттерны поведения	Интерпретатор (287)	Грамматика и интерпретация языка
	Итератор (302)	Способ перебора элементов агрегата
	Команда (275)	Время и способ выполнения запроса
	Наблюдатель (339)	Множество объектов, зависящих от другого объекта; способ, которым зависимые объекты поддерживают себя в актуальном состоянии
	Посетитель (379)	Операции, которые могут применяться к объекту или объектам, не меняя класса
	Посредник (319)	Взаимодействующие объекты и механизм их совместной работы
	Состояние (352)	Состояние объекта
	Стратегия (362)	Алгоритм
	Хранитель (330)	Закрытая информация, хранящаяся вне объекта, и время ее сохранения
	Цепочка обязанностей (263)	Объект, выполняющий запрос
	Шаблонный метод (373)	Шаги алгоритма

1.8. КАК ПОЛЬЗОВАТЬСЯ ПАТТЕРНОМ ПРОЕКТИРОВАНИЯ

Допустим, вы выбрали паттерн проектирования. Как теперь им пользоваться? Ниже описана последовательность действий, которая поможет вам эффективно применить паттерн:

1. *Прочитайте описание паттерна, чтобы получить о нем общее представление*. Особое внимание обратите на разделы «Применимость» и «Результаты» — убедитесь, что выбранный вами паттерн действительно подходит для решения ваших задач.
2. *Вернитесь назад и изучите разделы «Структура», «Участники» и «Отношения»*. Убедитесь, что вы понимаете упоминаемые в паттерне классы и объекты и то, как они взаимодействуют друг с другом.

3. *Просмотрите раздел «Пример кода» с конкретным примером применения паттерна в программе.* Изучение кода поможет понять, как нужно реализовывать паттерн.
4. *Выберите для участников паттерна подходящие имена.* Имена участников паттерна обычно слишком абстрактны, чтобы вставлять их непосредственно в код. Тем не менее бывает полезно включить имя участника как составную часть имени, используемого в программе. Это сделает факт применения паттерна более очевидным в реализации. Например, если вы пользуетесь паттерном стратегия в алгоритме размещения текста, то классы могли бы называться `SimpleLayoutStrategy` или `TeXLayoutStrategy`.
5. *Определите классы.* Объявите их интерфейсы, установите отношения наследования и определите переменные экземпляра, представляющие данные объекта и ссылки на другие объекты. Выявите в своем приложении классы, на которые паттерн оказывает влияние, и соответствующим образом модифицируйте их.
6. *Определите имена операций, встречающихся в паттерне.* Здесь, как и в предыдущем случае, имена обычно зависят от приложения. Руководствуйтесь теми функциями и взаимодействиями, которые ассоциированы с каждой операцией. Кроме того, будьте последовательны при выборе имен. Например, для обозначения фабричного метода можно было бы всюду использовать префикс `Create-`.
7. *Реализуйте операции, которые выполняют обязанности и обеспечивают взаимодействия, определенные в паттерне.* Советы о том, как это лучше сделать, вы найдете в разделе «Реализация». Пригодится и раздел «Пример кода».

Все вышесказанное — не более чем рекомендации. Со временем вы выработаете собственный подход к работе с паттернами проектирования.

Никакое обсуждение применения паттернов проектирования нельзя считать полным, если не сказать о том, как *не надо* их применять. Паттерны не должны применяться без разбора. Нередко за гибкость и простоту изменения, которые дают паттерны, приходится платить усложнением дизайна и/или ухудшением производительности. Паттерн проектирования стоит применять, только когда дополнительная гибкость действительно необходима. В оценке достоинств и недостатков паттерна большую помощь могут оказать разделы каталога «Результаты».

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].

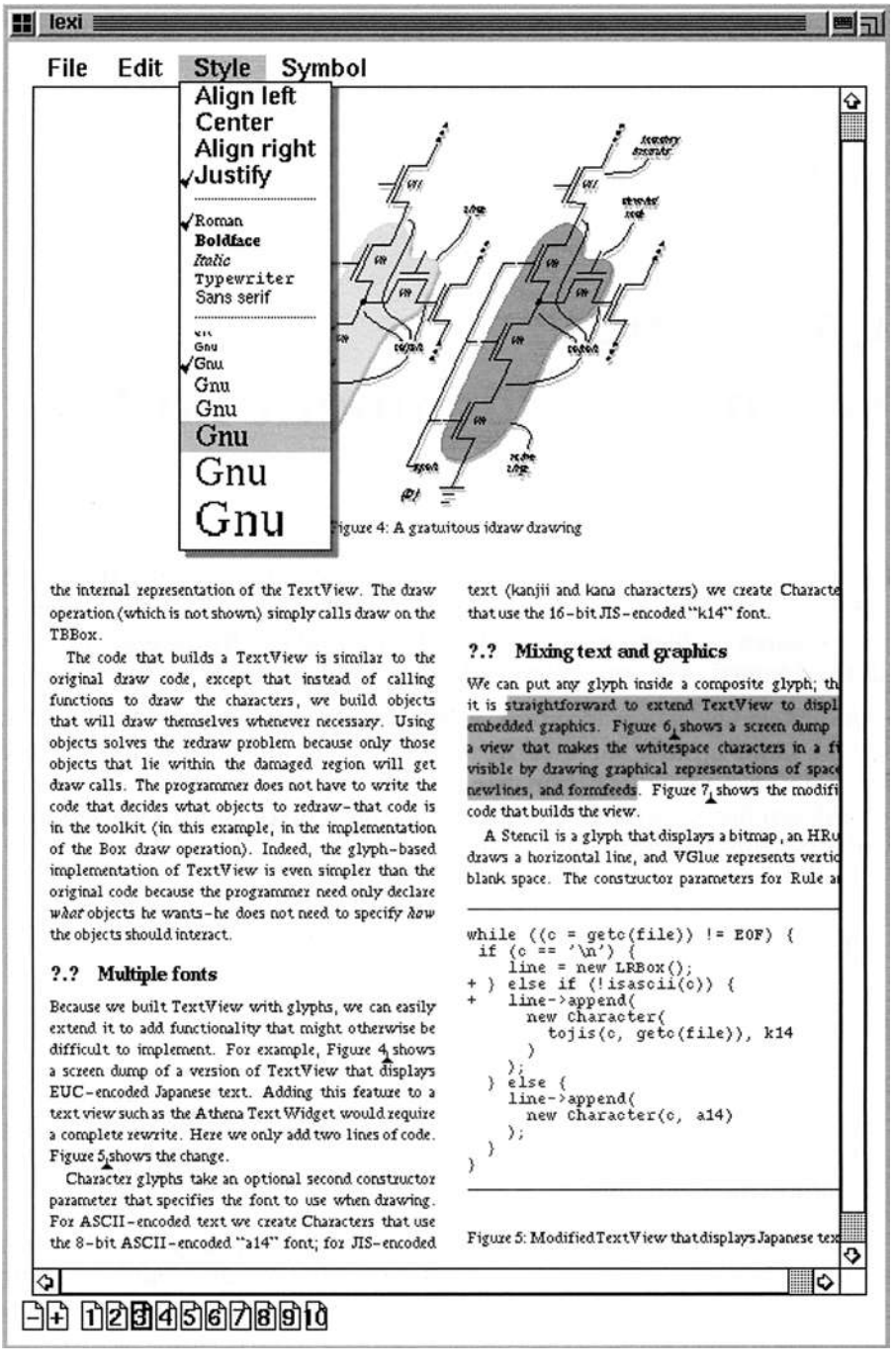


Рис. 2.1. Пользовательский интерфейс Lexi

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.

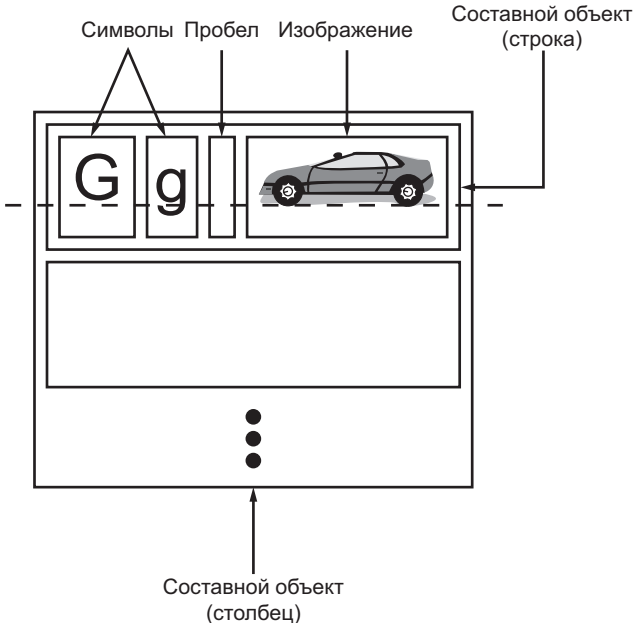


Рис. 2.2. Рекурсивная композиция текста и графики

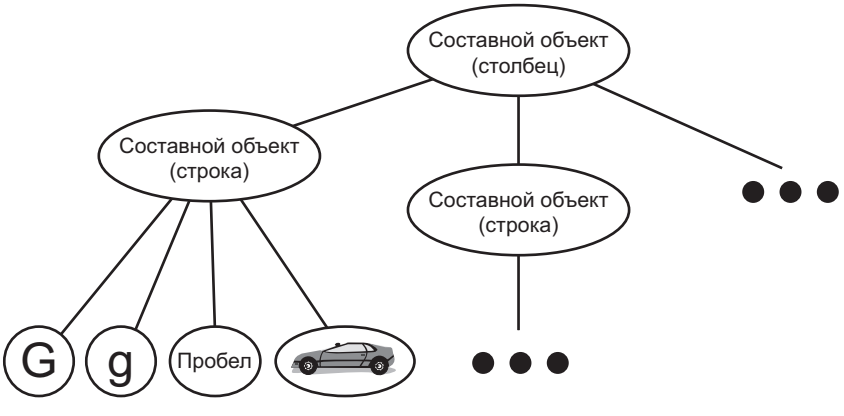


Рис. 2.3. Структура объекта для рекурсивной композиции текста и графики

У описанного подхода есть два важных следствия. Первое очевидно: для объектов нужны соответствующие классы. Второе, менее очевидное, состоит в том, что у этих классов должны быть совместимые интерфейсы, поскольку мы хотим унифицировать работу с ними. Для обеспечения совместимости интерфейсов в таком языке, как C++, применяется наследование.

ГЛИФЫ

Абстрактный класс `Glyph` (глиф) определяется для всех объектов, которые могут присутствовать в структуре документа[1]. Его подклассы определяют как примитивные графические элементы (скажем, символы и изображения), так и структурные элементы (строки и колонки). На рис. 2.4 изображена достаточно обширная часть иерархии класса `Glyph`, а в табл. 2.1 более подробно представлен базовый интерфейс этого класса в синтаксисе C++[2].

Таблица 2.1. Базовый интерфейс класса Glyph

Обязанность	Операции
Внешнее представление	virtual void Draw(Window*) virtual void Bounds(Rect&)
Обнаружение точки воздействия	virtual bool Intersects(const Point&)
Структура	virtual void Insert(Glyph*, int) virtual void Remove(Glyph*) virtual Glyph* Child(int) virtual Glyph* Parent()

У глифов есть три основные обязанности. Они (1) умеют рисовать себя на экране, (2) знают, сколько места они занимают, (3) располагают информацией о своих потомках и родителях.

Подклассы класса `Glyph` переопределяют операцию `Draw`, которая перерисовывает текущий объект в окне. При вызове `Draw` ей передается ссылка на объект `Window`. В классе `Window` определены графические операции для

---

[1] Впервые термин «глиф» в этом контексте употребил Пол Кальдер [CL90]. В большинстве современных редакторов документов отдельные символы не представляются объектами — вероятно, из соображений эффективности. Кальдер продемонстрировал практическую пригодность этого подхода в своей диссертации [Cal93]. Наши глифы проще предложенных им, поскольку мы для простоты ограничились строгими иерархиями. Глифы Кальдера могут использоваться совместно для уменьшения потребления памяти и образуют направленные ациклические графы. Для достижения того же эффекта можно воспользоваться паттерном приспособленец, но оставим это в качестве упражнения читателю.

[2] Представленный здесь интерфейс намеренно сделан минимальным, чтобы не загромождать обсуждение техническими деталями. Полный интерфейс должен включать операции для работы с графическими атрибутами: цветами, шрифтами и преобразованиями координат, а также операции для нетривиального управления потомками.

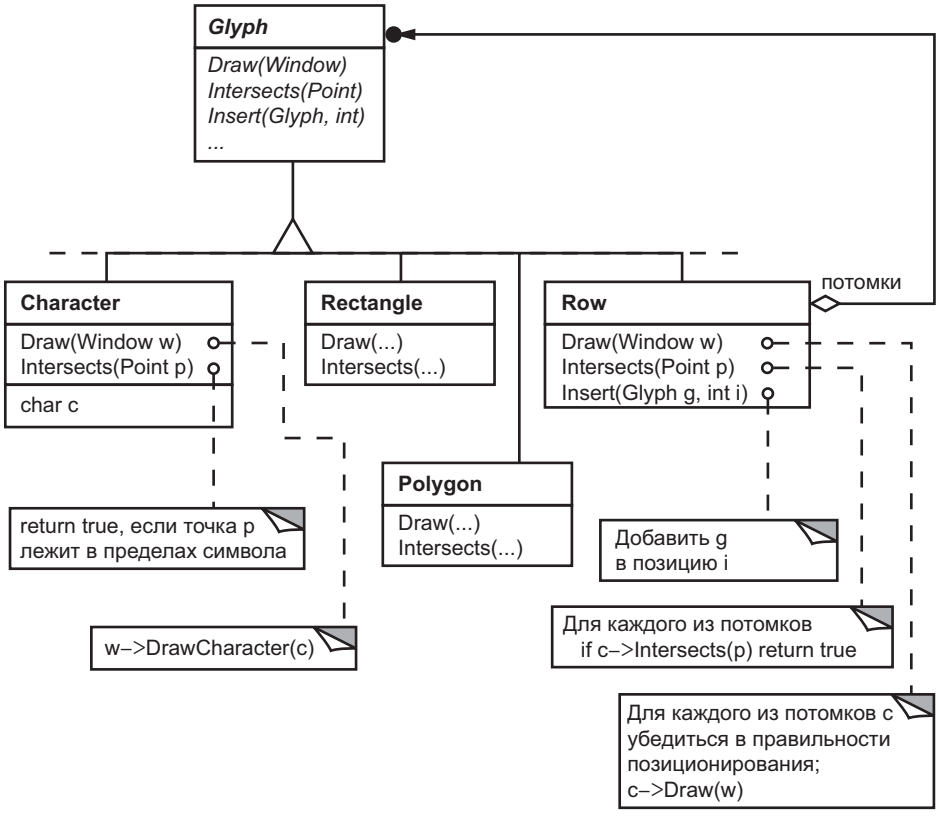


Рис. 2.4. Частичная иерархия класса Glyph

прорисовки в окне на экране текста и основных геометрических фигур. Например, в подклассе `Rectangle` операция `Draw` могла бы определяться так:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
```

Здесь `_x0`, `_y0`, `_x1` и `_y1` — переменные класса `Rectangle`, определяющие два противоположных угла прямоугольника, а `DrawRect` — операция класса `Window`, рисующая на экране прямоугольник.

Глифу-родителю часто бывает нужно знать, сколько места на экране занимает глиф-потомок — например, чтобы расположить его и остальные глифы в строке без перекрытий (как показано на рис. 2.3). Операция `Bounds` возвращает прямоугольную область, занимаемую глифом (точнее, противоположные углы наименьшего прямоугольника, содержащего глиф). В под-

классах класса `Glyph` эта операция переопределена так, чтобы она возвращала прямоугольную область, в которой осуществляется прорисовка.

Операция `Intersects` возвращает признак, показывающий, лежит ли заданная точка в пределах глифа. Всякий раз, когда пользователь щелкает мышью где-то в документе, Lexi вызывает эту операцию, чтобы определить, какой глиф или глифовая структура оказались под указателем мыши. Класс `Rectangle` переопределяет эту операцию для вычисления пересечения точки с прямоугольником.

Поскольку у глифов могут быть потомки, то нам необходим единый интерфейс для добавления, удаления и обхода потомков. Например, потомками класса `Row` являются глифы, расположенные в данной строке. Операция `Insert` вставляет глиф в позицию, заданную целочисленным индексом[1]. Операция `Remove` удаляет заданный глиф, если он действительно является потомком.

Операция `Child` возвращает потомка с заданным индексом (если таковой существует). Глифы, у которых действительно есть потомки (такие как `Row`), должны пользоваться операцией `Child`, а не обращаться к структуре данных потомка напрямую. В таком случае при изменении структуры данных, скажем, с массива на связанный список не придется модифицировать операции вроде `Draw`, которые перебирают всех потомков. Аналогично операция `Parent` предоставляет стандартный интерфейс для доступа к родителю глифа, если таковой имеется. В Lexi глифы хранят ссылку на своего родителя, а операция `Parent` просто возвращает эту ссылку.

ПАТТЕРН COMPOSITE (КОМПОНОВЩИК)

Рекурсивная композиция подходит не только для документов. Ей можно пользоваться для представления любых потенциально сложных иерархических структур. Паттерн компоновщик (196) инкапсулирует сущность рекурсивной композиции в объектно-ориентированных категориях. Сейчас самое время обратиться к разделу об этом паттерне и изучить его на примере только что рассмотренного сценария.

---

[1] Возможно, целочисленный индекс — не лучший способ описания потомков глифа. Это зависит от структуры данных, используемой внутри глифа. Если потомки хранятся в связанном списке, то более эффективно было бы передавать указатель на элемент списка. Более удачное решение проблемы индексации будет описано в разделе 2.8, когда будем обсуждать анализ документа.

2.3. ФОРМАТИРОВАНИЕ

Мы разобрались с тем, как *представлять* физическую структуру документа. Далее нужно разобраться с тем, как сконструировать *конкретную* физическую структуру, соответствующую правильно отформатированному документу. Представление и форматирование — это разные аспекты проектирования. По описанию внутренней структуры невозможно определить, как добраться до определенной подструктуры. За это в основном отвечает Lexi. Редактор разбивает текст на строки, строки — на колонки и т. д., учитывая при этом пожелания пользователя. Так, пользователь может изменить ширину полей, размеры отступов и позиций табуляции, установить одиночный или двойной междустрочный интервал, а также задать много других параметров форматирования[1]. Алгоритм форматирования Lexi должен все это учитывать.

Кстати говоря, мы ограничим значение термина «форматирование» и будем понимать под ним лишь разбиение на строки. Будем считать термины «форматирование» и «разбиение на строки» взаимозаменяемыми. Все приемы, рассматриваемые ниже, в равной мере относятся и к разбиению строк на колонки, и к разбиению колонок на страницы.

Таблица 2.2. Базовый интерфейс класса Compositor

Обязанность	Операции
Что форматировать	void SetComposition(Composition*)
Когда форматировать	virtual void Compose()

ИНКАПСУЛЯЦИЯ АЛГОРИТМА ФОРМАТИРОВАНИЯ

С учетом всех ограничений и многочисленных подробностей процесс форматирования с трудом поддается автоматизации. К этой проблеме есть много подходов, и было разработано много разных алгоритмов форматирования со

[1] Пользователя в большей степени интересует логическая структура документа: предложения, абзацы, разделы, главы и т. д. Физическая структура в общем-то менее интересна. Большинству пользователей не важно, где в абзаце произошел разрыв строки, если в целом все отформатировано правильно. То же самое относится и к форматированию колонок и страниц. Таким образом, пользователи задают только высокоуровневые ограничения на физическую структуру, а Lexi берет на себя всю черновую работу по их реализации.

своими сильными и слабыми сторонами. Поскольку Lexi — это WYSIWYG-редактор, важно выдержать баланс между качеством и скоростью форматирования. В общем случае желательно, чтобы редактор реагировал достаточно быстро и при этом внешний вид документа оставался приемлемым. На достижение этого компромисса влияет много факторов, и не все из них удастся установить на этапе компиляции. Например, можно предположить, что пользователь смирится с некоторым замедлением реакции в обмен на лучшее качество форматирования. При таком предположении следует применять совершенно другой алгоритм форматирования. Также возможен компромисс между временем и памятью, в большей степени ориентированный на реализацию: кэширование в памяти большего объема информации может уменьшить время форматирования.

Поскольку алгоритмы форматирования обычно оказываются весьма сложными, желательно, чтобы они были достаточно замкнутыми, а еще лучше — полностью независимыми от структуры документа. В оптимальном варианте добавление новой разновидности `Glyph` вовсе не затрагивает алгоритм форматирования. С другой стороны, при добавлении нового алгоритма форматирования не должно возникать необходимости в модификации существующих глифов.

Учитывая все вышесказанное, мы должны постараться спроектировать Lexi так, чтобы алгоритм форматирования можно было легко заменить по крайней мере на этапе компиляции, если уж не во время выполнения. Алгоритм можно изолировать и обеспечить возможность его простой замены путем инкапсуляции в объекте. А конкретнее мы определим отдельную иерархию классов для объектов, инкапсулирующих алгоритмы форматирования. Корнем иерархии станет интерфейс, который поддерживает широкий спектр алгоритмов, а каждый подкласс будет реализовывать этот интерфейс в виде конкретного алгоритма форматирования. Тогда удастся ввести подкласс класса `Glyph`, который будет автоматически структурировать своих потомков с помощью переданного ему объекта-алгоритма.

КЛАССЫ COMPOSITOR И COMPOSITION

Мы определим класс `Compositor` (композитор) для объектов, которые могут инкапсулировать алгоритм форматирования. Интерфейс (см. табл. 2.2) позволяет объекту этого класса узнать, *какие* глифы надо форматировать и *когда*. Форматируемые композитором глифы являются потомками специального подкласса класса `Glyph`, который называется `Composition` (композиция). Композиция при создании получает объект некоторого подкласса

`Compositor` (специализированный для конкретного алгоритма разбиения на строки) и в нужные моменты предписывает композитору строить композицию глифов по мере изменения документа пользователем. На рис. 2.5 изображены отношения между классами `Composition` и `Compositor`.

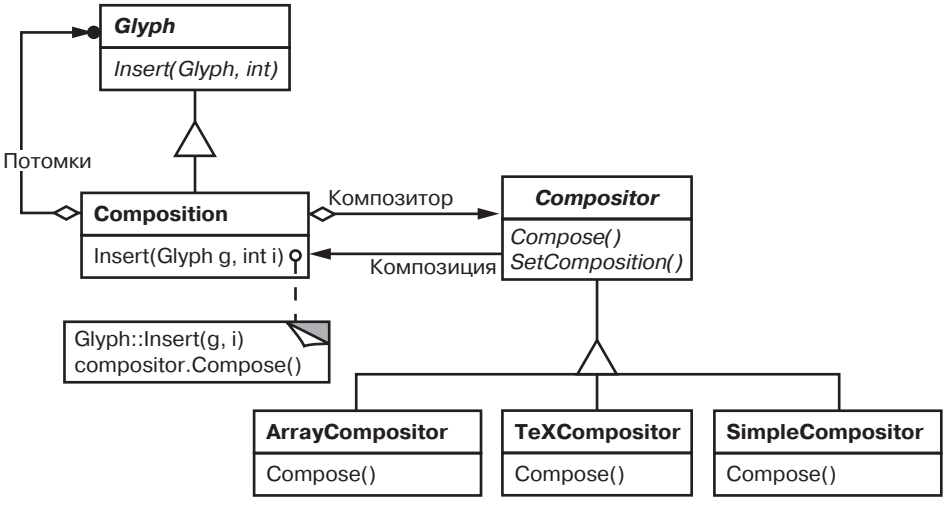


Рис. 2.5. Отношения классов Composition и Compositor

Неформатированный объект `Composition` содержит только видимые глифы, составляющие основное содержание документа. В нем нет глифов, определяющих физическую структуру документа, например `Row` и `Column`. В таком состоянии композиция находится сразу после создания и инициализации глифами, которые должна отформатировать. Во время форматирования композиция вызывает операцию `Compose` своего объекта `Compositor`. Композитор обходит всех потомков композиции и вставляет новые глифы `Row` и `Column` в соответствии со своим алгоритмом разбиения на строки[1]. На рис. 2.6 показана получающаяся объектная структура. Глифы, созданные и вставленные в эту структуру композитором, закрашены на рисунке серым цветом.

Каждый подкласс класса `Compositor` может реализовывать свой собственный алгоритм разбиения на строки. Например, класс `SimpleCompositor` мог бы осуществлять быстрый проход, не обращая внимания на такую экзотику, как

[1] Композитор должен получить коды символов глифов Character, чтобы вычислить места разбиения на строки. В разделе 2.8 мы увидим, как получить эту информацию полиморфно, не добавляя специфичной для символов операции к интерфейсу класса Glyph.

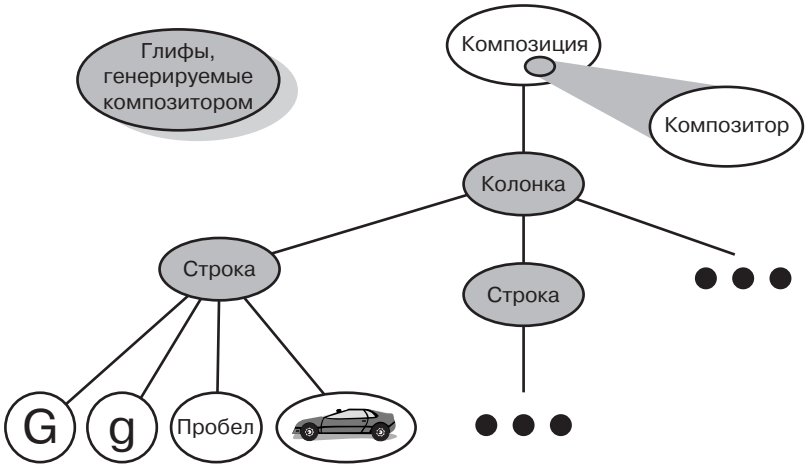


Рис. 2.6. Объектная структура, отражающая алгоритм разбиения на строки, выбираемый композитором

«цвет» документа. Под «хорошим цветом» понимается равномерное распределение текста и пустого пространства. Класс `TeXCompositor` мог бы реализовывать полный алгоритм TeX [Knu84], учитывающий наряду со многими другими вещами и цвет, но за счет увеличения времени форматирования.

Наличие классов `Compositor` и `Composition` позволяет отделить код, поддерживающий физическую структуру документа, от кода алгоритмов форматирования. Мы можем добавить новые подклассы для класса `Compositor`, не трогая классов глифов, и наоборот. Фактически для замены алгоритма разбиения на строки во время выполнения достаточно добавить единственную операцию `SetCompositor` к базовому интерфейсу класса `Composition`.

ПАТТЕРН STRATEGY (СТРАТЕГИЯ)

Инкапсуляция алгоритма в объект — это назначение паттерна стратегия (362). Основными участниками паттерна являются объекты-стратегии, инкапсулирующие различные алгоритмы, и контекст, в котором они работают. Композиторы представляют варианты стратегий; они инкапсулируют алгоритмы форматирования. Композиция образует контекст для стратегии композитора.

Ключ к применению паттерна стратегия — проектирование интерфейсов стратегии и контекста, достаточно общих для поддержки широкого диапазона

алгоритмов. Поддержка нового алгоритма не должна требовать изменения интерфейса стратегии или контекста. В нашем примере поддержка доступа к потомкам, их вставки и удаления, в базовом интерфейсе класса `Glyph` имеет достаточно общий характер, чтобы подклассы класса `Compositor` могли изменять физическую структуру документа независимо от того, с помощью каких алгоритмов это делается. Аналогичным образом интерфейс класса `Compositor` предоставляет композициям все, что им необходимо для запуска операции форматирования.

2.4. ОФОРМЛЕНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Рассмотрим два усовершенствования пользовательского интерфейса Lexi. Первое добавляет рамку вокруг области редактирования текста, чтобы четко обозначить границы страницы текста, второе — полосы прокрутки, при помощи которых пользователь просматривает разные части страницы. Чтобы упростить добавление и удаление таких элементов оформления (особенно во время выполнения), для их включения в пользовательский интерфейс не должно использоваться наследование. Максимальная гибкость достигается в том случае, если другим объектам пользовательского интерфейса даже не будет известно о том, какие еще элементы оформления в нем присутствуют. Это позволит добавлять и удалять декоративные элементы без изменения других классов.

ПРОЗРАЧНОЕ ОКРУЖЕНИЕ

В программировании улучшение пользовательского интерфейса подразумевает расширение существующего кода. Применение для этой цели наследования не дает возможности реорганизовать интерфейс во время выполнения. Не менее серьезной проблемой является комбинаторный рост числа классов в случае широкого использования наследования.

Можно было бы добавить рамку к классу `Composition`, породив от него новый подкласс `BorderedComposition`. Точно так же можно было бы добавить и интерфейс прокрутки, породив подкласс `ScrollableComposition`. Если же мы хотим иметь и рамку, и полосу прокрутки, следовало бы создать подкласс `BorderedScrollableComposition`, и так далее. Если довести эту идею до логического завершения, то пришлось бы создавать отдельный подкласс для каждой возможной комбинации декоративных элементов. Это решение быстро перестает работать с ростом количества таких декораций.

Композиция объектов предоставляет куда более приемлемый и гибкий механизм расширения. Но из каких объектов формировать композицию? Поскольку известно, что мы оформляем существующий глиф, то и сам элемент оформления могли бы сделать объектом (скажем, экземпляром класса `Border`). Следовательно, композиция может быть составлена из глифа и рамки. На следующем шаге необходимо решить, что во что включается. Можно считать, что рамка содержит глиф, и это разумно, так как рамка окружает глиф на экране. Можно принять и противоположное решение — поместить рамку внутрь глифа, но тогда пришлось бы модифицировать соответствующий подкласс класса `Glyph`, чтобы он «знал» о существовании рамки. Первый вариант — включение глифа в рамку — позволяет поместить весь код для отображения рамки в классе `Border`, оставив остальные классы без изменения.

Как выглядит класс `Border`? Тот факт, что у рамки есть визуальное представление, наталкивает на мысль, что она должна быть глифом, то есть подклассом класса `Glyph`. Но есть и более убедительные причины поступить именно так: клиентов не должно интересовать, есть у глифов рамки или нет. Все глифы должны обрабатываться единообразно. Когда клиент приказывает простому глифу без рамки нарисовать себя, тот делает это, не добавляя никаких элементов оформления. Если же этот глиф заключен в рамку, то клиент не должен как-то специально обрабатывать рамку; он просто предписывает составному глифу выполнить прорисовку точно так же, как и простому глифу в предыдущем случае. Отсюда следует, что интерфейс класса `Border` должен соответствовать интерфейсу класса `Glyph`. Чтобы гарантировать это, мы и делаем `Border` подклассом `Glyph`.

Все это подводит нас к идее *прозрачного окружения* (transparent enclosure), которая объединяет концепции: (1) композиции с одним потомком (однокомпонентные), и (2) совместимых интерфейсов. В общем случае клиенту неизвестно, имеет ли он дело с компонентом или его окружением (то есть родителем), особенно если окружение просто делегирует все операции своему единственному компоненту. Но окружение может также *расширять* поведение компонента, выполняя дополнительные действия либо до, либо после делегирования (а возможно, и до, и после). Окружение может также добавить компоненту состояние — как именно, будет показано ниже.

МОНОГЛИФ

Концепцию прозрачного окружения можно применить ко всем глифам, оформляющим другие глифы. Чтобы конкретизировать эту идею, определим подкласс класса `Glyph`, называемый `MonoGlyph`. Он будет выступать в роли аб-

страктного класса для глифов-декораций вроде рамки (см. рис. 2.7). В классе `MonoGlyph` хранится ссылка на компонент, которому он и переадресует все запросы. При этом `MonoGlyph` по определению становится абсолютно прозрачным для клиентов.

Вот как моноглиф реализует операцию `Draw`:

```
void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
```

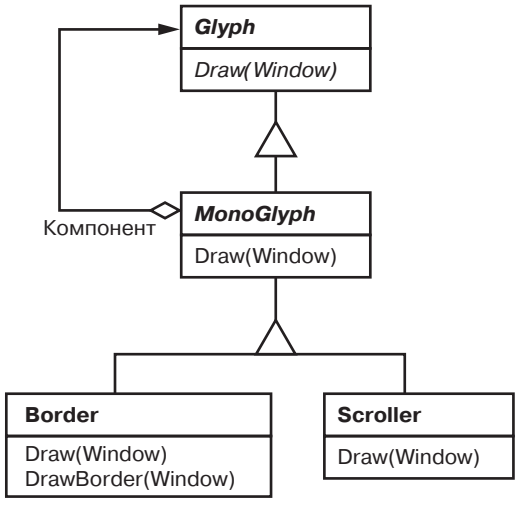


Рис. 2.7. Отношения класса MonoGlyph с другими классами

Подклассы `MonoGlyph` замещают по меньшей мере одну из таких операций переадресации. Например, `Border::Draw` сначала вызывает операцию родительского класса `MonoGlyph::Draw`, чтобы компонент выполнил свою часть работы, то есть нарисовал все, кроме рамки. Затем `Border::Draw` рисует рамку вызовом своей закрытой операции `DrawBorder`, детали которой мы опустим:

```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

Обратите внимание, что `Border::Draw`, по сути дела, *расширяет* операцию родительского класса, чтобы нарисовать рамку. Это не то же самое, что простая замена операции: в таком случае `MonoGlyph::Draw` не вызывалась бы.

На рис. 2.7 показан другой подкласс класса `MonoGlyph`. `Scroller` — это `MonoGlyph`, который рисует свои компоненты на экране в зависимости от положения двух полос прокрутки, добавляющихся в качестве элементов оформления. Когда `Scroller` отображает свой компонент, он приказывает своей графической системе обрезать его по границам окна. Отсеченные части компонента, оказавшиеся за пределами видимой части окна, не появляются на экране.

Теперь у нас есть все, что необходимо для добавления рамки и интерфейса прокрутки к области редактирования текста в Lexi. Мы помещаем имеющийся экземпляр класса `Composition` в экземпляр класса `Scroller`, чтобы добавить интерфейс прокрутки, а результат композиции еще раз погружаем в экземпляр класса `Border`. Получившийся объект показан на рис. 2.8.

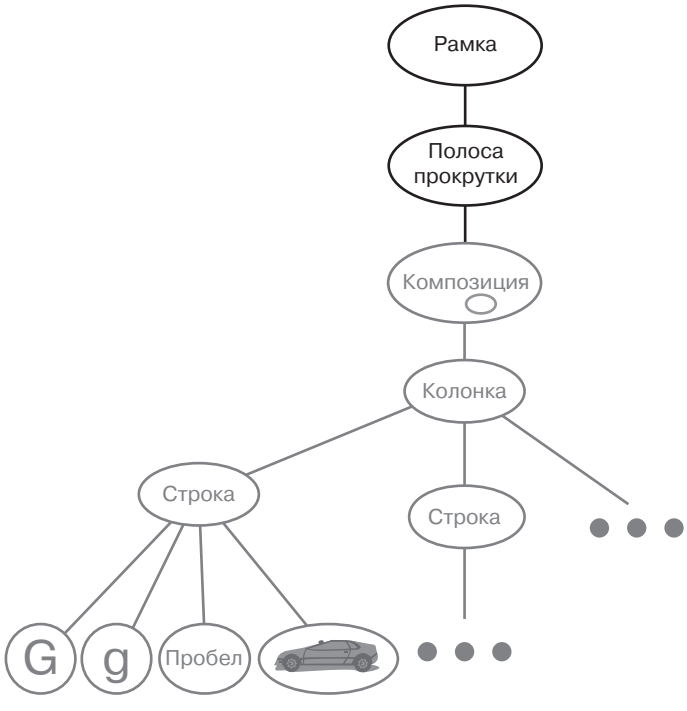


Рис. 2.8. Объектная структура после добавления элементов оформления

Обратите внимание, что с таким же успехом можно было бы использовать обратный порядок композиции, сначала добавив рамку, а потом поместив результат в `Scroller`. В таком случае рамка прокручивалась бы вместе с текстом. Может быть, это именно то, что вам нужно, а может, и нет. Здесь важно то, что прозрачное окружение легко позволяет клиенту экспериментировать с разными вариантами без знания подробностей кода, добавляющего декорации.

Отметим, что рамка допускает композицию не более чем с одним глифом. Этим она отличается от рассмотренных выше композиций, где родительскому объекту позволялось иметь сколько угодно потомков. Здесь же заключение чего-то в рамку предполагает, что это «что-то» имеется в единственном экземпляре. Мы могли бы приписать некоторую семантику декорации более одного объекта, но тогда пришлось бы вводить множество видов композиций с оформлением: оформление строки, колонки и т. д. Это не улучшит архитектуру, так как у нас уже есть классы для такого рода композиций. Поэтому для композиции лучше использовать уже существующие классы, а новые добавлять для оформления результата. Отделение декорации от других видов композиции одновременно упрощает классы, реализующие разные элементы оформления, и уменьшает их количество. Кроме того, мы избавляемся от необходимости дублировать уже имеющуюся функциональность.

ПАТТЕРН DECORATOR (ДЕКОРАТОР)

Паттерн декоратор (209) абстрагирует отношения между классами и объектами, необходимые для поддержки оформления с помощью техники прозрачного окружения. Термин «оформление» на самом деле применяется в более широком смысле, чем мы видели выше. В паттерне декоратор под ним понимается нечто, расширяющее круг обязанностей объекта. Можно, например, представить себе оформление абстрактного дерева синтаксического разбора семантическими действиями, конечного автомата — новыми состояниями или сети, состоящей из устойчивых объектов, — тегами атрибутов. Декоратор обобщает подход, который мы использовали в Lexi, чтобы расширить его область применения.

2.5. ПОДДЕРЖКА НЕСКОЛЬКИХ СТАНДАРТОВ ОФОРМЛЕНИЯ

При проектировании системы приходится сталкиваться с проблемой переносимости между различными программными и аппаратными платформами. Перенос Lexi на другую платформу не должен требовать капитального перепроектирования, иначе не стоит за него и браться. Он должен быть максимально прост.

Одним из препятствий для переноса является разнообразие стандартов оформления, призванных унифицировать работу с приложениями на данной платформе. Эти стандарты определяют, как приложения должны выглядеть и реагировать на действия пользователя. Хотя существующие стандарты не так уж сильно отличаются друг от друга, ни один пользователь не спутает

один стандарт с другим — приложения для Motif выглядят не совсем так, как аналогичные приложения на других платформах, и наоборот. Программа, работающая более чем на одной платформе, на всех платформах должна соответствовать принятой стилистике пользовательского интерфейса.

Одна из целей проектирования — сделать так, чтобы Lexi поддерживал разные стандарты внешнего облика и чтобы легко можно было добавить поддержку нового стандарта сразу же после его появления (а это неизбежно произойдет). Хотелось бы также, чтобы наш дизайн решал и другую задачу: изменение оформления Lexi во время выполнения.

АБСТРАГИРОВАНИЕ СОЗДАНИЯ ОБЪЕКТА

Все, что мы видим и с чем можем взаимодействовать в пользовательском интерфейсе Lexi, — это визуальные глифы, скомпонованные в другие, уже невидимые глифы вроде строки (`Row`) и колонки (`Column`). Невидимые глифы объединяют видимые — скажем, кнопку (`Button`) или символ (`Character`) — и правильно располагают их на экране. В стилевых руководствах много говорится о внешнем облике и поведении так называемых «виджетов» (widgets); это просто другое название таких видимых глифов, как кнопки, полосы прокрутки и меню, выполняющих в пользовательском интерфейсе функции элементов управления. Для представления данных виджеты могут пользоваться более простыми глифами: символами, окружностями, прямоугольниками и многоугольниками.

Будем считать, что классы глифов-виджетов, с помощью которых реализуются стандарты оформления, делятся на два класса:

- набор абстрактных подклассов класса `Glyph` для каждой категории виджетов. Например, абстрактный класс `ScrollBar` будет дополнять интерфейс глифа с целью получения операций прокрутки общего вида, а `Button` — это абстрактный класс, добавляющий операции с кнопками;
- набор конкретных подклассов для каждого абстрактного подкласса, в которых реализованы стандарты внешнего облика. Так, у `ScrollBar` могут быть подклассы `MotifScrollBar` и `PMScrollBar`, реализующие полосы прокрутки в стиле Motif и Presentation Manager соответственно.

Lexi должен различать глифы-виджеты для разных стилей внешнего оформления. Например, когда необходимо поместить в интерфейс кнопку, редактор должен создать экземпляр подкласса класса `Glyph` для нужного стиля кнопки (`MotifButton`, `PMButton`, `MacButton` и т. д.).

Ясно, что в реализации Lexi это нельзя сделать непосредственно — например, вызовом конструктора, если речь идет о языке C++. При этом была бы жестко запрограммирована кнопка одного конкретного стиля, а значит, выбрать нужный стиль во время выполнения оказалось бы невозможно. Кроме того, мы были бы вынуждены отслеживать и изменять каждый такой вызов конструктора при переносе Lexi на другую платформу. А ведь кнопки — это лишь один элемент пользовательского интерфейса Lexi. Загромождение кода вызовами конструкторов для разных классов оформления создает существенные неудобства при сопровождении. Стоит что-нибудь пропустить — и в приложении для Mac появится меню в стиле Motif.

Lexi необходимо каким-то образом определить нужный стандарт оформления для создания подходящих виджетов. При этом надо не только постараться избежать явных вызовов конструкторов, но и уметь без труда заменять весь набор виджетов. Этого можно добиться путем *абстрагирования процесса создания объекта*. Следующий пример пояснит, что имеется в виду.

ФАБРИКИ И ИЗГОТОВЛЕННЫЕ КЛАССЫ

В обычном случае экземпляр глифа полосы прокрутки в стиле Motif создается следующим кодом на C++:

```
ScrollBar* sb = new MotifScrollBar;
```

Но если вы хотите свести к минимуму зависимость Lexi от стандарта оформления, именно такого кода следует избегать. Предположим, однако, что `sb` инициализируется так:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

где `guiFactory` — экземпляр класса `MotifFactory`. Операция `CreateScrollBar()` возвращает новый экземпляр подходящего подкласса `ScrollBar`, который соответствует нужному варианту оформления, в данном случае Motif. С точки зрения клиентов результат тот же, что и при прямом вызове конструктора `MotifScrollBar`. Но есть и существенное отличие: нигде в коде больше не упоминается имя Motif. Объект `guiFactory` абстрагирует процесс создания полос прокрутки не только для Motif, но и для *любых* стандартов оформления. Более того, `guiFactory` не ограничивается изготовлением только полос прокрутки и может применяться для производства любых виджетов, включая кнопки, поля ввода, меню и т. д.

Все это возможно благодаря тому, что `MotifFactory` является подклассом `GUIFactory` — абстрактного класса, который определяет общий интерфейс для

создания глифов-виджетов. В нем есть такие операции, как `CreateScrollBar` и `CreateButton`, для создания экземпляров различных видов виджетов. Подклассы `GUIFactory` реализуют эти операции, возвращая глифы вроде `MotifScrollBar` и `PMButton`, реализующие конкретное оформление и поведение. На рис. 2.9 показана иерархия классов для объектов `GUIFactory`.

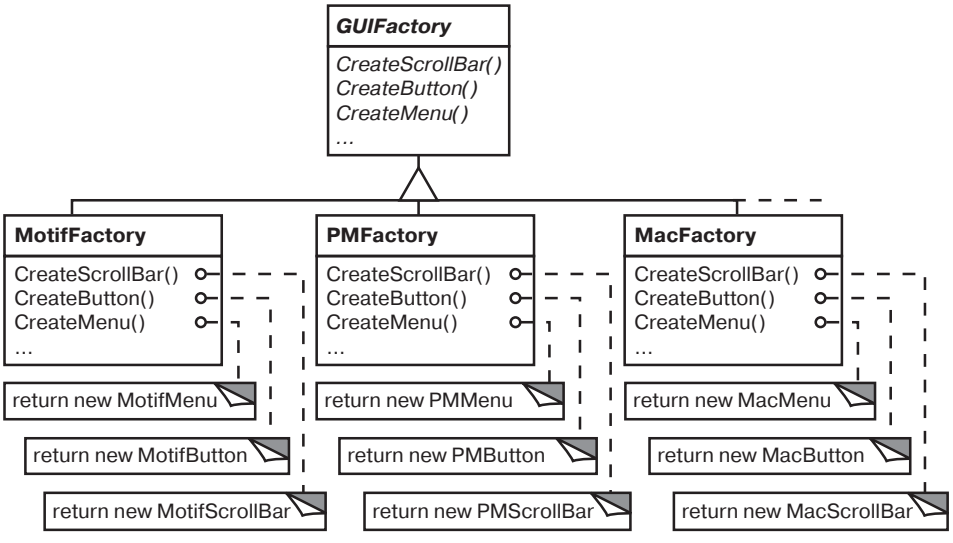


Рис. 2.9. Иерархия классов GUIFactory

Мы говорим, что фабрики *изготавливают* объекты. Все объекты, изготовленные на фабриках, связаны друг с другом; в нашем случае все такие продукты — это виджеты, имеющие один и тот же внешний облик. На рис. 2.10 показаны некоторые классы, необходимые для того, чтобы фабрика могла изготавливать глифы-виджеты.

Остается ответить на последний вопрос: как получить экземпляр `GUIFactory`? Да как угодно, лишь бы это было удобно. Переменная `guiFactory` может быть глобальной, может быть статическим членом хорошо известного класса или даже локальной, если весь пользовательский интерфейс создается внутри одного класса или функции. Существует специальный паттерн проектирования одиночка (157), предназначенный для работы с такого рода объектами, существующими в единственном экземпляре. Важно, однако, чтобы фабрика `guiFactory` была инициализирована *до того*, как начнет использоваться для производства объектов, но *после того*, как стало известно, какое оформление требуется.

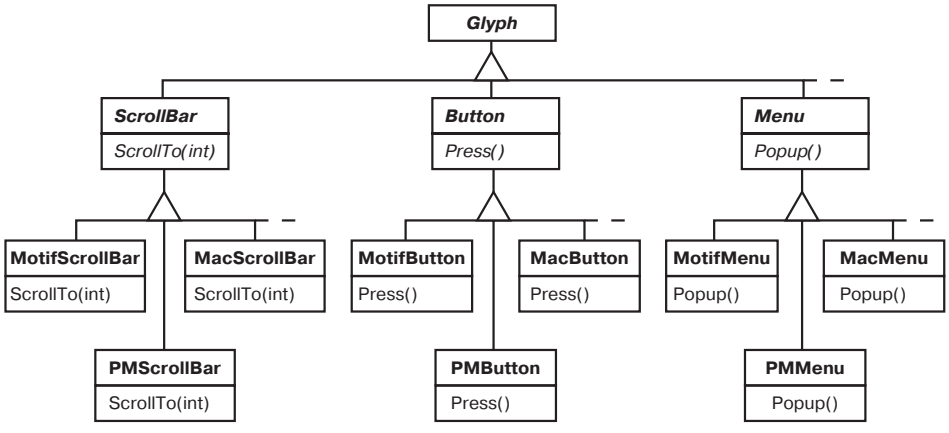


Рис. 2.10. Абстрактные классы-продукты и их конкретные подклассы

Когда вариант оформления известен на этапе компиляции, то `guiFactory` можно инициализировать простым присваиванием в начале программы:

```
GUIFactory* guiFactory = new MotifFactory;
```

Если же пользователь может задать нужный вариант оформления с помощью строки-параметра при запуске, то код создания фабрики мог бы выглядеть так:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
    // Задается пользователем или средой при запуске
if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;
} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;
} else {
    guiFactory = new DefaultGUIFactory;

}
```

Существуют и более сложные способы выбора фабрики во время выполнения. Например, можно было бы вести реестр, в котором символьные строки ассоциируются с объектами фабрик. Это позволяет зарегистрировать экземпляр новой фабрики без изменения существующего кода, как требуется при предыдущем подходе. И вам не придется связывать с приложением код фабрик для всех конкретных платформ. Это существенно, поскольку связать код для `MotifFactory` с приложением, работающим на платформе, где Motif не поддерживается, может оказаться невозможным.

Впрочем, важно лишь то, что после настройки приложения для работы с конкретной фабрикой объектов, мы получаем нужный вариант оформления. Если впоследствии мы изменим решение, то сможем инициализировать `guiFactory` по-другому, чтобы изменить внешний облик, а затем динамически перестроим интерфейс. Независимо от того, когда и как будет инициализироваться `guiFactory`, можно быть уверенным в том, что после этого приложение сможет создать необходимый вариант оформления без каких-либо изменений.

ПАТТЕРН ABSTRACT FACTORY (АБСТРАКТНАЯ ФАБРИКА)

Фабрики и их продукция — ключевые участники паттерна абстрактная фабрика (113). Этот паттерн может создавать семейства объектов без явного создания экземпляров. Применять его лучше всего тогда, когда число и общий вид изготавливаемых объектов остаются постоянными, но между конкретными семействами продуктов имеются различия. Выбор того или иного семейства осуществляется путем создания экземпляра конкретной фабрики, после чего она используется для создания всех объектов. Подставив вместо одной фабрики другую, можно заменить все семейство объектов целиком. В паттерне абстрактная фабрика акцент делается на создании *семейств* объектов, и это отличает его от других порождающих паттернов, создающих только один какой-то вид объектов.

2.6. ПОДДЕРЖКА НЕСКОЛЬКИХ ОКОННЫХ СИСТЕМ

Как должно выглядеть приложение — это лишь один из многих вопросов, встающих при переносе приложения на другую платформу. Еще одна проблема из той же серии — оконная среда, в которой работает Lexi. Данная среда создает иллюзию наличия нескольких перекрывающихся окон на одном растровом дисплее. Она распределяет между окнами площадь экрана и направляет им события клавиатуры и мыши. Сегодня существует несколько широко распространенных и во многом не совместимых между собой оконных систем (например, Macintosh, Presentation Manager, Windows, X). Мы хотели бы, чтобы Lexi работал в любой оконной среде по тем же причинам, по которым мы поддерживаем несколько стандартов оформления.

МОЖНО ЛИ ВОСПОЛЬЗОВАТЬСЯ АБСТРАКТНОЙ ФАБРИКОЙ?

На первый взгляд представляется, что и в этом случае можно воспользоваться паттерном абстрактная фабрика. Но ограничения, связанные с переносом на

другие оконные системы, существенно отличаются от тех, что накладывают независимость от оформления.

Применяя паттерн абстрактная фабрика, мы предполагали, что удастся определить конкретный класс глифов-виджетов для каждого стандарта оформления. Это означало, что можно будет произвести конкретный класс для конкретного стандарта (например, `MotifScrollBar` и `MacScrollBar`) от абстрактного класса (допустим, `ScrollBar`). Предположим, однако, что у нас уже есть несколько иерархий классов, полученных от разных поставщиков, — по одной для каждого стандарта. Крайне маловероятно, что данные иерархии будут совместимы между собой. Поэтому в приложении не будет общих абстрактных изготавливаемых классов для каждого вида виджетов (`ScrollBar`, `Button`, `Menu` и т. д.) — а без них фабрика классов работать не может. Необходимо, чтобы иерархии виджетов имели единый набор абстрактных интерфейсов. Только тогда удастся правильно объявить операции `Create`... в интерфейсе абстрактной фабрики.

Для виджетов эта проблема была решена разработкой собственных абстрактных и конкретных изготавливаемых классов. Теперь аналогичная трудность возникает при попытке заставить Lexi работать во всех существующих оконных средах; а именно, разные среды имеют несовместимые интерфейсы программирования. Но на этот раз все сложнее, поскольку мы не можем себе позволить реализовать собственную нестандартную оконную систему.

Однако спасительный выход все же есть. Как и стандарты оформления, интерфейсы оконных систем не так уж радикально отличаются друг от друга, ибо все они предназначены примерно для одних и тех же целей. Нам нужен унифицированный набор оконных абстракций, которым было бы возможно закрыть любую конкретную реализацию оконной системы.

ИНКАПСУЛЯЦИЯ ЗАВИСИМОСТЕЙ ОТ РЕАЛИЗАЦИИ

В разделе 2.2 был введен класс `Window` для отображения на экране глифа или структуры, состоящей из глифов. Ничего не говорилось о том, с какой оконной системой работает этот объект, поскольку в действительности он вообще не связан ни с одной системой. Класс `Window` инкапсулирует функциональность окна в любой оконной системе:

- операции прорисовки базовых геометрических фигур;
- возможность свернуть и развернуть окно;
- изменение собственных размеров;

- перерисовка своего содержимого при необходимости — например, при развертывании из значка или открытии ранее перекрытой части окна.

Класс `Window` должен охватывать функциональность окон из разных оконных систем. Рассмотрим два крайних подхода:

- *пересечение функциональности*. Интерфейс класса `Window` предоставляет только функциональность, общую для всех оконных систем. Однако в результате мы получаем интерфейс не богаче, чем в самой слабой из рассматриваемых систем. Мы не можем воспользоваться более мощными средствами, даже если их поддерживает большинство оконных систем (но не все);
- *объединение функциональности*. Создается интерфейс, который включает возможности *всех* существующих систем. Здесь возникает опасность получить чрезмерно громоздкий и внутренне противоречивый интерфейс. Кроме того, нам придется изменять его (а вместе с ним и Lexi) всякий раз, как только производитель переработает интерфейс своей оконной системы.

Ни одно из крайних решений не годится, поэтому мы выберем компромиссное. Класс `Window` будет предоставлять удобный интерфейс, поддерживающий наиболее популярные возможности оконных систем. Поскольку редактор Lexi будет работать с классом `Window` напрямую, этот класс должен поддерживать и сущности, о которых Lexi известно — то есть глифы. Это означает, что интерфейс класса `Window` должен включать базовый набор графических операций, позволяющий глифам отображать себя в окне. В табл. 2.3 приведена подборка операций из интерфейса класса `Window`.

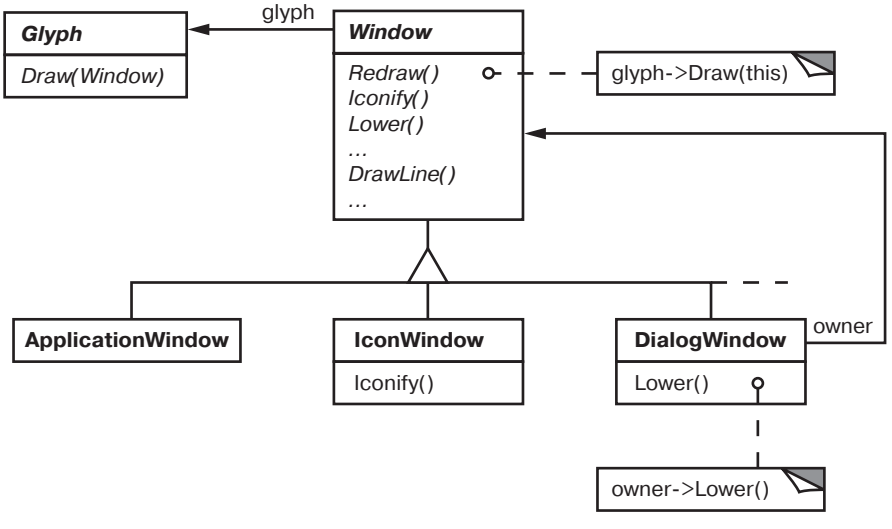
Таблица 2.3. Интерфейс класса Window

Обязательный	Операции

Управление окнами	`virtual void Redraw()`
	`virtual void Raise()`
	`virtual void Lower()`
	`virtual void Iconify()`
	`virtual void Deiconify()`
	`...`

Графика	`virtual void DrawLine(...)`
	`virtual void DrawRect(...)`
	`virtual void DrawPolygon(...)`
	`virtual void DrawText(...)`
	`...`

`Window` — это абстрактный класс. Его конкретные подклассы поддерживают различные виды окон, с которыми имеет дело пользователь. Например, окна приложений, сообщений, значки — это все окна, но свойства у них разные. Для учета таких различий мы можем определить подклассы `ApplicationWindow`, `IconWindow` и `DialogWindow`. Возникающая иерархия позволяет таким приложениям, как Lexi, создать унифицированную, интуитивно понятную абстракцию окна, не зависящую от оконной системы конкретного поставщика:



Итак, мы определили оконный интерфейс, с которым будет работать Lexi. Но где же в нем место для реальной платформеннозависимой оконной системы? Если мы не собираемся реализовывать собственную оконную систему, то в каком-то месте наша абстракция окна должна быть выражена в терминах целевой системы. Но где именно?

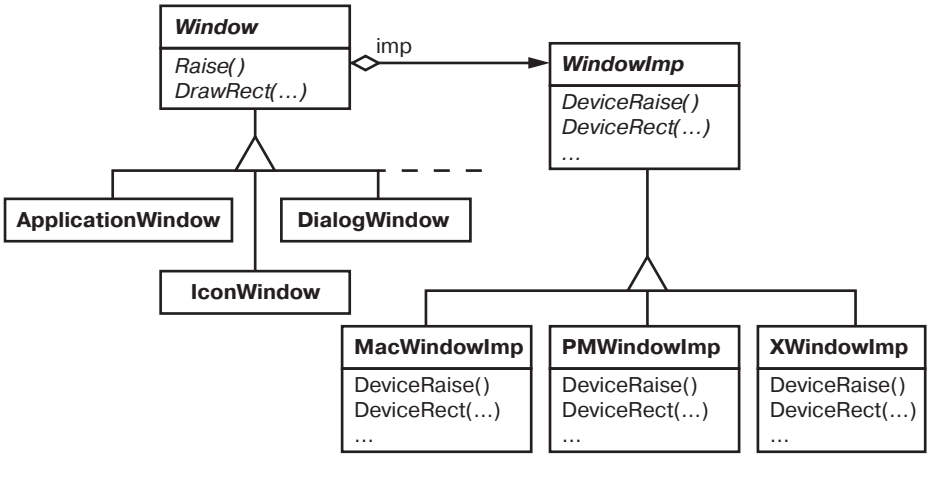
Можно было бы реализовать несколько вариантов класса `Window` и его подклассов — по одному для каждой оконной среды. Выбор нужного варианта производится при сборке Lexi для данной платформы. Но представьте себе, с чем вы столкнетесь при сопровождении, если придется отслеживать множество разных классов с одним и тем же именем `Window`, но реализованных для разных оконных систем. Вместо этого можно было бы создать зависящие от реализации подклассы каждого класса в иерархии `Window`, но закончилось бы это тем же самым стремительным ростом числа классов, о котором уже говорилось при попытке добавить элементы оформления. Кроме того, оба решения не обладают достаточной гибкостью,

чтобы можно было перейти на другую оконную систему уже после компиляции программы. Поэтому придется поддерживать несколько разных исполняемых файлов.

Ни тот, ни другой вариант не вдохновляют, но что еще можно сделать? То же самое, что мы сделали для форматирования и декорирования, — *инкапсулировать изменяющуюся сущность*. В этом случае переменной частью является реализация оконной системы. Если инкапсулировать функциональность оконной системы в объекте, то удастся реализовать свой класс `Window` и его подклассы в категориях интерфейса этого объекта. Более того, если такой интерфейс сможет поддерживать все интересующие нас оконные системы, то не придется изменять ни `Window`, ни его подклассы при переходе на другую систему. Чтобы настроить оконные объекты в соответствии с требованиями нужной оконной системы, достаточно передать им подходящий объект, инкапсулирующий оконную систему. Это можно сделать даже во время выполнения.

КЛАССЫ WINDOW И WINDOWIMP

Мы определим отдельную иерархию классов `WindowImp`, в которой скроем знание о различных реализациях оконных систем. `WindowImp` — это абстрактный класс для объектов, инкапсулирующих системнозависимый код. Чтобы заставить Lexi работать в конкретной оконной системе, каждый оконный объект будем конфигурировать экземпляром того подкласса `WindowImp`, который предназначен для этой системы. На схеме ниже представлены отношения между иерархиями `Window` и `WindowImp`:



Скрыв реализацию в классах `WindowImp`, мы сумели избежать «засорения» классов `Window` зависимостями от оконной системы. В результате иерархия `Window` получается сравнительно компактной и стабильной. В то же время мы можем расширить иерархию реализаций, если будет нужно поддержать новую оконную систему.

ПОДКЛАССЫ WINDOWIMP

Подклассы `WindowImp` преобразуют запросы в операции, характерные для конкретной оконной системы. Рассмотрим пример из раздела 2.2. Мы определили `Rectangle::Draw` в категориях `DrawRect` над экземпляром класса `Window`:

```
void Rectangle::Draw (Window* w) {
    w->DrawRect(_x0, _y0, _x1, _y1);
}
```

В реализации `DrawRect` по умолчанию используется абстрактная операция рисования прямоугольников, объявленная в `WindowImp`:

```
void Window::DrawRect ( Coord x0, Coord y0, Coord x1, Coord y1    )
{    _imp->DeviceRect(x0, y0, x1, y1);    }
```

где `_imp` — переменная класса `Window`, в которой хранится указатель на объект `WindowImp`, использованный при настройке `Window`. Реализация окна определяется тем экземпляром подкласса `WindowImp`, на который указывает `_imp`. Для `XWindowImp` (то есть подкласса `WindowImp` для оконной системы X Window System) реализация `DeviceRect` могла бы выглядеть так:

```
void XWindowImp::DeviceRect ( Coord x0, Coord y0, Coord x1, Coord y1 )
{
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

`DeviceRect` определяется именно так, поскольку `XDrawRectangle` (интерфейс X Window для рисования прямоугольников) определяет прямоугольник по левому нижнему углу, ширине и высоте. Реализация `DeviceRect` должна вычислить эти значения по переданным ей параметрам. Сначала она находит левый нижний угол (поскольку (x0, y0) может быть любым из четырех углов прямоугольника), а затем вычисляет длину и ширину.

PMWindowImp (подкласс WindowImp для Presentation Manager) определил бы DeviceRect по-другому:

```
void PMWindowImp::DeviceRect ( Coord x0, Coord y0, Coord x1, Coord y1 )
{
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];
    point[0].x = left; point[0].y = top;
    point[1].x = right;  point[1].y = top;
    point[2].x = right;  point[2].y = bottom;
    point[3].x = left; point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR)  ||
        (GpiEndPath(_hps) == false) )
    {
        // Сообщить об ошибке
    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Откуда такое отличие от версии для X? Дело в том, что в Presentation Manager (PM) нет явной операции для рисования прямоугольников, как в X. Вместо этого PM имеет более общий интерфейс для задания вершин фигуры, состоящей из нескольких отрезков (множество таких вершин называется *траекторией*), и для рисования границы или заливки той области, которую эти отрезки ограничивают.

Очевидно, что реализации DeviceRect для PM и X совершенно непохожи, но это не имеет никакого значения. Возможно, WindowImp скрывает различия интерфейсов оконных систем за большим, но стабильным интерфейсом. Это позволяет автору подкласса Window сосредоточиться на абстракции окна, а не на подробностях оконной системы. Также появляется возможность добавлять поддержку для новых оконных систем, не изменяя классы из иерархии Window.

НАСТРОЙКА КЛАССА WINDOW С ПОМОЩЬЮ WINDOWIMP

Важнейший вопрос, который мы еще не рассмотрели, — как настроить окно подходящим подклассом WindowImp? Другими словами, когда инициализиру-

ется переменная _imp и как узнать, какая оконная система (а следовательно, и подкласс WindowImp) используется? Чтобы окно могло сделать что-то нетривиальное, ему необходим объект WindowImp.

Есть несколько возможностей, но мы остановимся на той, где используется паттерн абстрактная фабрика (113). Можно определить абстрактный фабричный класс WindowSystemFactory, предоставляющий интерфейс для создания различных видов объектов в зависимости от оконной системы:

```
class WindowSystemFactory {
public:
    virtual WindowImp* CreateWindowImp() = 0;
    virtual ColorImp* CreateColorImp() = 0;
    virtual FontImp* CreateFontImp() = 0;

    // Операция "Create..." для всех ресурсов оконной системы
};
```

Далее можно определить конкретную фабрику для каждой оконной системы:

```
class PMWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
        { return new PMWindowImp; }
    // ...
};

class XWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
        { return new XWindowImp; }
    // ...
};
```

Для инициализации переменной _imp указателем на объект WindowImp, соответствующий данной оконной системе, конструктор базового класса Window может использовать интерфейс WindowSystemFactory:

```
Window::Window () {
    _imp = windowSystemFactory->CreateWindowImp();
}
```

Переменная windowSystemFactory — это известный программе экземпляр подкласса WindowSystemFactory (аналог переменной guiFactory, определяющей вариант оформления). И инициализируется переменная windowSystemFactory точно так же.

ПАТТЕРН BRIDGE (МОСТ)

Класс `WindowImp` определяет интерфейс к общим средствам оконной системы, но на его дизайн накладываются иные ограничения, нежели на интерфейс класса `Window`. Прикладной программист не обращается к интерфейсу `WindowImp` непосредственно, он имеет дело только с объектами класса `Window`. Поэтому интерфейс `WindowImp` необязательно должен соответствовать представлению программиста о мире, как то было в случае с иерархией и интерфейсом класса `Window`. Интерфейс `WindowImp` может более точно отражать сущности, которые в действительности предоставляют оконные системы, со всеми их особенностями. Он может быть ориентирован на пересечение или объединение функциональности — в зависимости от того, что лучше подходит для целевой оконной системы.

Важно понимать, что интерфейс класса `Window` призван обслуживать интересы прикладного программиста, тогда как интерфейс класса `WindowImp` в большей степени ориентирован на оконные системы. Разделение функциональности окон между иерархиями `Window` и `WindowImp` позволяет независимо реализовывать и специализировать их интерфейсы. Объекты из этих иерархий взаимодействуют, позволяя Lexi работать без изменений в нескольких оконных системах.

Отношение иерархий `Window` и `WindowImp` являет собой пример паттерна мост (184). Его идея заключается в том, чтобы иерархии классов могли работать совместно даже в случае, если они эволюционировали по отдельности. Критерии разработки, которыми мы руководствовались, заставили нас создать две различные иерархии классов: одну, поддерживающую логическую концепцию окон, и другую для воплощения разных реализаций окон. Паттерн мост позволяет сохранять и совершенствовать логические абстракции управления окнами, не прикасаясь к коду, зависящему от оконной системы, и наоборот.

2.7. ОПЕРАЦИИ ПОЛЬЗОВАТЕЛЯ

Часть функциональности Lexi доступна через WYSIWYG-представление документа. Вы вводите и удаляете текст, перемещаете точку вставки и выбираете фрагменты текста, просто указывая и щелкая мышью или нажимая клавиши. Другая часть функциональности доступна через выпадающие меню, кнопки и горячие клавиши. В частности, к этой категории относятся следующие операции:

- создание нового документа;
- открытие, сохранение и печать существующего документа;
- вырезание выбранной части документа и вставка ее в другое место;
- изменение шрифта и стиля выбранного текста;
- изменение форматирования текста (например, установка режима выравнивания);
- завершение работы приложения и др.

Lexi предоставляет для этих операций различные пользовательские интерфейсы. Но мы не хотим ассоциировать конкретную операцию с определенным пользовательским интерфейсом, поскольку для выполнения одной и той же операции желательно иметь несколько интерфейсов (например, листать страницы можно как помощью кнопки на экране, так и командой меню). Кроме того, отдельные элементы интерфейса могут измениться в будущем.

Кроме того, эти операции реализуются в разных классах. Нам как разработчикам хотелось бы иметь доступ к функциональности классов, не создавая зависимостей между классами реализации и пользовательского интерфейса. В противном случае получится сильно связанный код, который будет трудно понять, расширять и сопровождать.

Ситуация осложняется еще и тем, что Lexi должен поддерживать операции отмены и повтора[1] большинства, *но не всех* операций. Точнее, желательно уметь отменять операции модификации документа (скажем, удаление), которые из-за оплошности пользователя могут привести к уничтожению большого объема данных. Но не следует пытаться отменить такую операцию, как сохранение чертежа или завершение приложения. Мы также не хотели бы налагать произвольные ограничения на число уровней отмены и повтора.

Разумеется, поддержка пользовательских операций распределена по всему приложению. Задача в том, чтобы найти простой и расширяемый механизм, удовлетворяющий всем вышеизложенным требованиям.

ИНКАПСУЛЯЦИЯ ЗАПРОСА

С точки зрения проектировщика выпадающее меню — это просто еще один вид вложения глифов. От других глифов, имеющих потомков, его отличает

[1] Под повтором (redo) понимается выполнение только что отмененной операции.

то, что большинство содержащихся в меню глифов каким-то образом реагирует на отпускание кнопки мыши.

Предположим, что такие реагирующие глифы являются экземплярами подкласса `MenuItem` класса `Glyph` и что свою работу они выполняют в ответ на запрос клиента[1]. Для выполнения запроса может потребоваться вызвать одну операцию одного объекта или много операций разных объектов (или какой-нибудь промежуточный вариант).

Нам не хватает механизма параметризации пунктов меню запросами, которые они должны выполнять. Таким способом удалось бы избежать разрастания числа подклассов и обеспечить большую гибкость во время выполнения. `MenuItem` можно параметризовать вызываемой функцией, но это решение неполно по трем причинам:

- в нем не учитывается проблема отмены/повтора;
- с функцией трудно ассоциировать состояние. Например, функция, изменяющая шрифт, должна знать, *какой именно* это шрифт;
- функции трудно расширять, а повторное использование их частей затруднено.

Отсюда следует, что пункты меню лучше параметризовать *объектом*, а не функцией. Тогда мы сможем прибегнуть к механизму наследования для расширения и повторного использования реализации запроса. Кроме того, у нас появляется место для сохранения состояния и реализации отмены/повтора. Это еще один пример инкапсуляции изменяющейся сущности, в данном случае — запроса. Каждый запрос мы инкапсулируем в объект-команду.

КЛАСС COMMAND И ЕГО ПОДКЛАССЫ

Сначала определим абстрактный класс `Command`, который будет предоставлять интерфейс для выдачи запроса. Базовый интерфейс включает всего одну абстрактную операцию `Execute`. Подклассы `Command` по-разному реализуют эту операцию для выполнения разных запросов. Некоторые подклассы могут частично или полностью делегировать работу другим объектам, а остальные выполняют запрос сами (рис. 2.11). Однако для запрашиваю-

[1] Концептуально клиентом является пользователь Lexi, но на самом деле это просто какой-то другой объект (например, диспетчер событий), который управляет обработкой ввода пользователя.

щего объект Command — это всего лишь объект Command; все такие объекты обрабатываются одинаково.

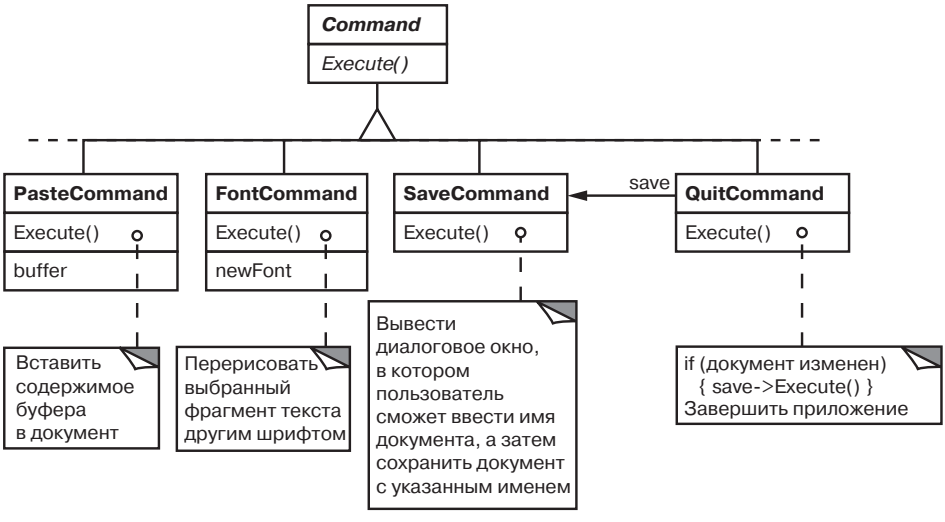


Рис. 2.11. Часть иерархии класса Command

Теперь в классе MenuItem может храниться объект, инкапсулирующий запрос (рис. 2.12). Каждому объекту, представляющему пункт меню, передается экземпляр того из подклассов Command, который соответствует этому пункту, точно так же, как мы задаем текст, отображаемый в пункте меню. Когда пользователь выбирает некоторый пункт меню, объект MenuItem просто вызывает операцию Execute для своего объекта Command, чтобы он выполнил запрос. Заметим, что кнопки и другие виджеты могут пользоваться объектами Command точно так же, как и пункты меню.

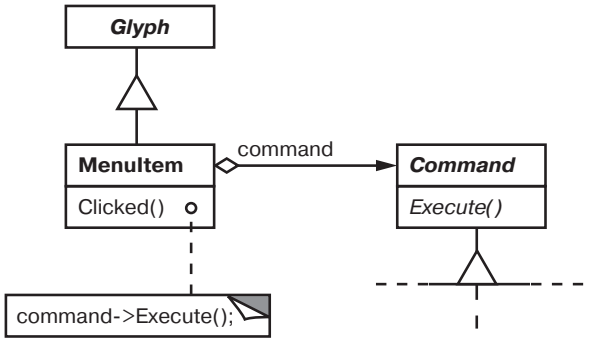


Рис. 2.12. Отношения между классами MenuItem и Command

ОТМЕНА ОПЕРАЦИЙ

Функциональность отмены/повтора играет важную роль в интерактивных приложениях. Чтобы иметь возможность отменять и повторять команды, нужно включить операцию `Unexecute` в интерфейс класса `Command`. Ее выполнение отменяет все последствия предыдущей операции `Execute` с использованием информации, сохраненной этой операцией. Так, при команде `FontCommand` операция `Execute` должна была бы сохранить диапазон текста, шрифт которого был изменен, а также первоначальный шрифт (или шрифты). Операция `Unexecute` класса `FontCommand` восстановит старый шрифт (или шрифты) для указанного диапазона текста.

Иногда возможность выполнения отмены должна определяться во время выполнения. Скажем, запрос на изменение шрифта выделенного участка текста не производит никаких действий, если текст уже отображен требуемым шрифтом. Предположим, что пользователь выбрал некий текст и решил изменить его шрифт на случайно выбранный. Что произойдет в результате последующего запроса на отмену? Должно ли бессмысленное изменение приводить к столь же бессмысленной отмене? Наверное, нет. Если пользователь повторит случайное изменение шрифта несколько раз, то не следует заставлять его выполнять точно такое же число отмен, чтобы вернуться к последнему осмысленному состоянию. Если суммарный эффект выполнения последовательности команд нулевой, то нет необходимости вообще делать что-либо при запросе на отмену.

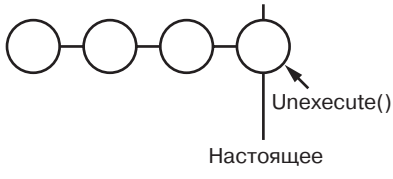
Для определения того, можно ли отменить действие команды, мы добавим к интерфейсу класса `Command` абстрактную операцию `Reversible` (обратимая), которая возвращает булево значение. Подклассы могут переопределить эту операцию и возвращать `true` или `false` в зависимости от критерия, вычисляемого во время выполнения.

ИСТОРИЯ КОМАНД

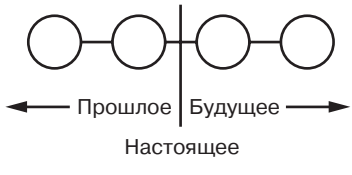
Последний шаг по поддержке отмены и повтора с произвольным числом уровней — определение истории команд, то есть списка ранее выполненных или отмененных команд. С концептуальной точки зрения история команд выглядит так:

Каждый кружок представляет один объект `Command`. В данном случае пользователь выполнил четыре команды. Первой была выполнена крайняя левая команда, затем вторая слева и т. д. вплоть до последней команды (крайней правой). Линия с пометкой «настоящее» обозначает самую последнюю выполненную (или отмененную) команду.

Чтобы отменить последнюю команду, мы просто вызываем операцию `Unexecute` для самой последней команды:

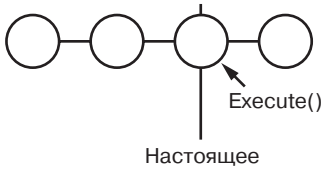


После отмены команды сдвигаем линию «настоящее» на одну команду влево. Если пользователь выполнит еще одну отмену, то произойдет откат еще на один шаг (см. рис. ниже).

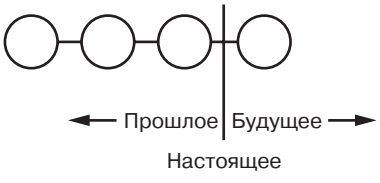


Видно, что за счет простого повторения процедуры мы получаем произвольное число уровней отмены, ограниченное лишь длиной списка истории команд.

Чтобы повторить только что отмененную команду, проведем обратные действия. Команды справа от линии «настоящее» — те, что могут быть повторены в будущем. Для повтора последней отмененной команды мы вызываем операцию `Execute` для последней команды справа от линии «настоящее»:



Затем линия «настоящее» сдвигается, чтобы следующий повтор вызвал операцию `Execute` для следующей команды будущего.



Разумеется, если следующая операция — это не повтор, а отмена, то команда слева от линии «настоящее» будет отменена. Таким образом, пользователь может перемещаться в обоих направлениях, чтобы исправить ошибки.

ПАТТЕРН COMMAND (КОМАНДА)

Команды Lexi — это пример применения паттерна команда (275), который описывает инкапсуляцию запроса. Этот паттерн предписывает единый интерфейс для выдачи запросов, с помощью которого можно настроить клиентов для обработки разных запросов. Интерфейс изолирует клиента от реализации запроса. Команда может полностью или частично делегировать реализацию запроса другим объектам либо выполнять данную операцию самостоятельно. Это идеальное решение для приложений типа Lexi, которые должны предоставлять централизованный доступ к функциональности, разбросанной по разным частям программы. Данный паттерн предлагает также механизмы отмены и повтора, построенные на основе базового интерфейса класса `Command`.

2.8. ПРОВЕРКА ПРАВОПИСАНИЯ И РАССТАНОВКА ПЕРЕНОСОВ

Последняя задача проектирования связана с анализом текста, а конкретно с проверкой правописания и нахождением мест, где можно поставить перенос для улучшения форматирования.

Ограничения здесь аналогичны тем, о которых уже говорилось при обсуждении форматирования в разделе 2.3. Как и в случае с разбиением на строки, существует много возможных реализаций поиска орфографических

ошибок и вычисления точек переноса. Поэтому и здесь планировалась поддержка нескольких алгоритмов. Пользователь сможет выбрать тот алгоритм, который его больше устраивает по соотношению затрат памяти, скорости и качества. Добавление новых алгоритмов тоже должно реализовываться просто.

Также необходимо избежать жесткой привязки этой информации к структуре документа. В данном случае такая цель даже более важна, чем при форматировании, поскольку проверка правописания и расстановка переносов — лишь два вида анализа текста, которые Lexi мог бы поддерживать. Со временем мы неизбежно захотим расширить аналитические возможности Lexi. Мы могли бы добавить поиск, подсчет слов, средства вычислений для суммирования значений в таблице, проверку грамматики и т. д. Но мы не хотим изменять класс `Glyph` и все его подклассы при каждом добавлении такого рода функциональности.

У этой задачи есть два аспекта: (1) доступ к анализируемой информации, разбросанной по разным глифам в структуре документа, и (2) собственно проведение анализа. Рассмотрим их по отдельности.

ДОСТУП К РАСПРЕДЕЛЕННОЙ ИНФОРМАЦИИ

Для многих видов анализа необходимо обрабатывать текст на уровне отдельных символов. Но анализируемый текст рассеян по иерархии структур, состоящих из объектов-глифов. Для анализа текста, представленного в таком виде, понадобится механизм доступа, располагающий информацией о структурах данных, в которых хранится текст. У одних глифов потомки могут храниться в связанных списках, у других — в массивах, а у третьих и вовсе используются какие-то экзотические структуры. Наш механизм доступа должен справляться со всем этим.

К сожалению, для разных видов анализа методы доступа к информации могут различаться. Обычно текст сканируется от начала к концу. Но иногда требуется сделать прямо противоположное. Например, для обратного поиска нужно проходить по тексту в обратном, а не в прямом направлении. А при вычислении алгебраических выражений может потребоваться симметричный (in order) обход.

Итак, наш механизм доступа должен уметь адаптироваться к разным структурам данных и поддерживать разные способы обхода (например, обход в прямом или обратном порядке или симметричный обход).

ИНКАПСУЛЯЦИЯ ДОСТУПА И ПОРЯДКА ОБХОДА

Пока что в нашем интерфейсе глифов для обращения к потомкам со стороны клиентов используется целочисленный индекс. Может, это и будет эффективно для тех классов глифов, которые хранят потомков в массиве, но совершенно неэффективно для глифов, использующих связанный список. Абстракция глифов должна скрыть структуру данных, в которой хранятся потомки. Тогда мы сможем изменить структуру данных, используемую классом глифа, не затрагивая другие классы.

Поэтому только глиф может знать, какую структуру он использует. Отсюда следует, что интерфейс глифов не должен отдавать предпочтение какой-то одной структуре данных. Например, не следует оптимизировать его в пользу массивов, а не связанных списков, как это делалось до сих пор.

Мы можем решить проблему и одновременно поддержать несколько разных способов обхода. Разумно включить множественные средства обращения и обхода прямо в классы глифов и предоставить способ выбирать между ними — например, с передачей константы из некоторого перечисления. Выполняя обход, классы передают этот параметр друг другу, чтобы гарантировать, что все они обходят структуру в одном и том же порядке. Так же должна передаваться любая информация, собранная во время обхода.

Для поддержки описанного подхода в интерфейс класса `Glyph` можно было бы добавить следующие абстрактные операции:

```
void First(Traversal kind)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

Операции `First`, `Next` и `IsDone` управляют обходом. `First` инициализирует процедуру обхода. В параметре ей передается разновидность обхода в виде константы из перечисления `Traversal`, которая может принимать такие значения, как `CHILDREN` (обходить только прямых потомков глифа), `PREORDER` (обходить всю структуру в прямом порядке), `POSTORDER` (в обратном порядке) или `INORDER` (в симметричном порядке). `Next` переходит к следующему глифу в порядке обхода, а `IsDone` сообщает, завершился ли обход. `GetCurrent` заменяет операцию `Child` — осуществляет доступ к текущему в данном обходе глифу. Старая операция `Insert` заменяется, теперь она вставляет глиф в текущую позицию. При анализе можно было бы использовать следующий код C++ для обхода структуры глифов с корнем `g` в прямом порядке:

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();

    // Выполнить анализ
}
```

Обратите внимание: целочисленный индекс исключен из интерфейса глифов. Не осталось ничего, что предполагало бы какой-то предпочтительный контейнер. Также клиенты были бы избавлены от необходимости самостоятельно реализовывать типичные виды доступа.

Но этот подход еще не идеален. Во-первых, он не позволяет поддерживать новые виды обхода без расширения множества значений перечисления или добавления новых операций. Предположим, вам нужен вариант прямого обхода, при котором автоматически пропускаются нетекстовые глифы. Тогда пришлось бы изменить перечисление `Traversal` и включить в него значение вида `TEXTUAL_PREORDER`.

Тем не менее, менять уже имеющиеся объявления нежелательно. Помещение всего механизма обхода в иерархию класса `Glyph` затрудняет модификацию и расширение без изменения многих других классов. Также затруднено повторное использование этого механизма для обхода других видов структур объектов. Наконец, у данной структуры не может быть более одного незавершенного обхода.

И снова наилучшее решение — инкапсуляция изменяющейся сущности в классе. В нашем случае это механизмы обращения к элементам и обхода. Вомзожно ввести класс объектов, называемых *итераторами*, единственное назначение которых — определить разные наборы таких механизмов. Можно также воспользоваться наследованием для унификации доступа к разным структурам данных и поддержки новых видов обхода. Тогда вам не придется изменять интерфейсы глифов или трогать реализации существующих глифов.

КЛАСС ITERATOR И ЕГО ПОДКЛАССЫ

Мы применим абстрактный класс `Iterator` для определения общего интерфейса обращения к элементам и обхода. Конкретные подклассы (такие как `ArrayIterator` и `ListIterator`) реализуют данный интерфейс для предоставления доступа к массивам и спискам, а такие подклассы, как `PreorderIterator`, `PostorderIterator` и им подобные, реализуют разные

виды обхода структур. Каждый подкласс класса `Iterator` содержит ссылку на структуру, которую он обходит. Экземпляры подкласса инициализируются этой ссылкой при создании. На рис. 2.13 показан класс `Iterator` и некоторые из его подклассов. Обратите внимание: в интерфейс класса `Glyph` добавлена абстрактная операция `CreateIterator` для поддержки итераторов.

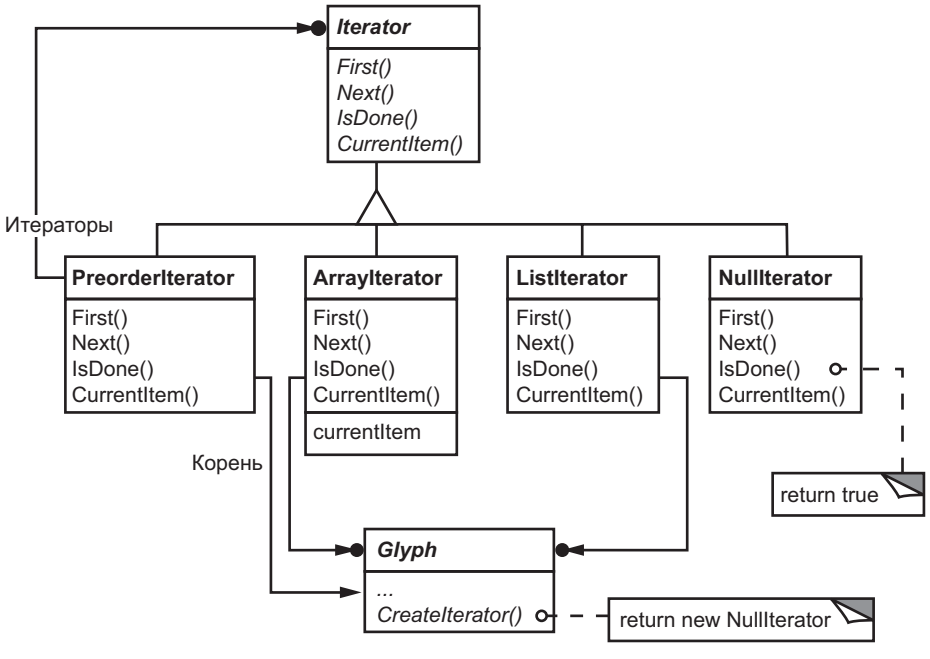


Рис. 2.13. Класс Iterator и его подклассы

Интерфейс итератора предоставляет операции `First`, `Next` и `IsDone` для управления обходом. В классе `ListIterator` операция `First` реализуется указателем на первый элемент списка, а `Next` перемещает итератор к следующему элементу. Операция `IsDone` возвращает признак, говорящий о том, перешел ли указатель за последний элемент списка. Операция `CurrentItem` разыменовывает итератор для возвращения глифа, на который он ссылается. Класс `ArrayIterator` делает то же самое с массивами глифов.

Теперь мы можем обращаться к потомкам в структуре глифа, не зная ее представления:

```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();
```

```
for (i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();

  // Выполнить действие с текущим потомком
}
```

`CreateIterator` по умолчанию возвращает экземпляр `NullIterator` — вырожденный итератор для глифов, у которых нет потомков, то есть листовых глифов. Операция `IsDone` для `NullIterator` всегда возвращает `true`.

Подкласс глифа, имеющего потомков, замещает операцию `CreateIterator` так, что она возвращает экземпляр другого подкласса класса `Iterator`. *Какого именно* — зависит от структуры, в которой содержатся потомки. Если подкласс `Row` класса `Glyph` размещает потомков в списке, то его операция `CreateIterator` будет выглядеть примерно так:

```
Iterator<Glyph*>* Row::CreateIterator () {
    return new ListIterator<Glyph*>(_children);
}
```

Итераторы для обхода в прямом и симметричном порядке реализуют алгоритм обхода в контексте конкретных глифов. В обоих случаях итератору передается корневой глиф той структуры, которую нужно обойти. Итераторы вызывают `CreateIterator` для каждого глифа в этой структуре и сохраняют возвращенные итераторы в стеке.

Например, класс `PreorderIterator` получает итератор от корневого глифа, инициализирует его так, чтобы он указывал на свой первый элемент, а затем помещает в стек:

```
void PreorderIterator::First () {
    Iterator<Glyph*>* i = _root->CreateIterator();
    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```

`CurrentItem` будет просто вызывать операцию `CurrentItem` для итератора на вершине стека:

```
Glyph* PreorderIterator::CurrentItem () const {
    Return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

Операция `Next` получает итератор с вершины стека и приказывает его текущему элементу создать свой итератор, спускаясь тем самым по структуре глифов как можно ниже (как это делается для прямого порядка). `Next` устанавливает новый итератор так, чтобы он указывал на первый элемент в порядке обхода, и помещает его в стек. Затем `Next` проверяет последний встретившийся итератор; если его операция `IsDone` возвращает `true`, значит, обход текущего поддерева (или листа) закончен. В таком случае `Next` снимает итератор с вершины стека и повторяет всю последовательность действий, пока не найдет следующее неполностью обойденное дерево, если таковое существует. Если же необойденных деревьев больше нет, то обход завершен:

```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();

    i->First();
    _iterators.Push(i);

    while (
        _iterators.Size() > 0 && _iterators.Top()->IsDone()
    ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

Обратите внимание: класс `Iterator` позволяет вводить новые виды обходов, не изменяя классы глифов, — достаточно породить новый подкласс и добавить новый обход так, как было сделано для `PreorderIterator`. Подклассы класса `Glyph` используют тот же самый интерфейс, чтобы предоставить клиентам доступ к своим потомкам без раскрытия внутренней структуры данных, в которой они хранятся. Поскольку итераторы сохраняют собственную копию состояния обхода, то одновременно можно иметь несколько активных итераторов для одной и той же структуры. И, хотя в нашем примере мы занимались обходом структур глифов, ничто не мешает параметризовать класс типа `PreorderIterator` типом объекта структуры. В C++ для этого использовались бы шаблоны. Тогда описанный механизм итераторов можно было бы применить для обхода других структур.

ПАТТЕРН ITERATOR (ИТЕРАТОР)

Паттерн итератор (302) абстрагирует описанный метод поддержки обхода структур, состоящих из объектов, и доступа к их элементам. Он применим

не только к составным структурам, но и к группам, абстрагирует алгоритм обхода и изолирует клиентов от деталей внутренней структуры объектов, которые они обходят. Паттерн итератор — еще один пример того, как инкапсуляция изменяющейся сущности помогает добиться гибкости и повторной используемости. Но все равно проблема итерации оказывается неожиданно глубокой, поэтому паттерн итератор гораздо сложней, чем было рассмотрено выше.

ОБХОД И ДЕЙСТВИЯ, ВЫПОЛНЯЕМЫЕ ПРИ ОБХОДЕ

Итак, теперь, когда у нас есть способ обойти структуру глифов, нужно заняться проверкой правописания и расстановкой переносов. Для обоих видов анализа необходимо накапливать собранную во время обхода информацию.

Прежде всего следует решить, на какую часть программы возложить ответственность за выполнение анализа. Можно было бы поручить это классам `Iterator`, тем самым сделав анализ неотъемлемой частью обхода. Но решение стало бы более гибким и пригодным для повторного использования, если бы обход был отделен от действий, которые при этом выполняются. Дело в том, что для одного и того же вида обхода могут выполняться разные виды анализа. Поэтому один и тот же набор итераторов можно было бы использовать для разных аналитических операций. Например, прямой порядок обхода применяется в разных случаях, включая проверку правописания, расстановку переносов, поиск в прямом направлении и подсчет слов.

Итак, анализ и обход следует разделить. На кого еще можно возложить ответственность за выполнение анализа? Мы знаем, что разновидностей анализа достаточно много. При каждом виде анализа в определенные моменты обхода будут выполняться разные действия. В зависимости от вида анализа некоторые глифы могут оказаться более важными, чем другие. При проверке правописания и расстановке переносов следует рассматривать только символьные глифы и пропускать графические — линии, растровые изображения и т. д. Если мы занимаемся разделением цветов, то желательно было бы ограничиться только видимыми глифами. Таким образом, разные виды анализа будут просматривать разные глифы.

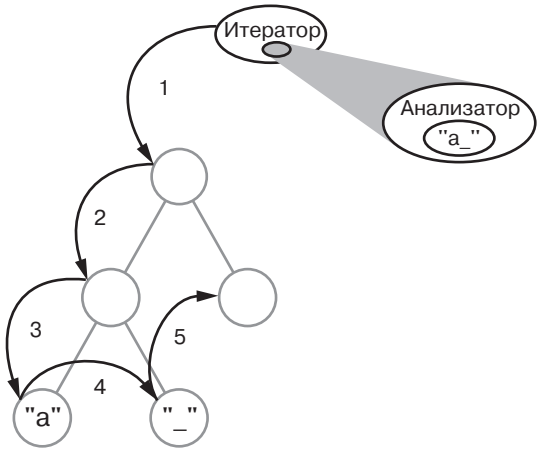
Поэтому данный вид анализа должен уметь различать глифы по их типу. Очевидное решение — встроить аналитическую функциональность в сами классы глифов. Тогда для каждого вида анализа мы можем добавить одну или несколько абстрактных операций в класс `Glyph` и реализовать их в подклассах в соответствии с той ролью, которую они играют при анализе.

Однако у такого подхода есть и недостаток: каждый класс глифов придется изменять при добавлении нового вида анализа. В некоторых случаях проблему удается сгладить: если в анализе участвует немного классов или если большинство из них выполняют анализ одним и тем же способом, то можно поместить подразумеваемую реализацию абстрактной операции прямо в класс `Glyph`. Такая операция по умолчанию будет обрабатывать наиболее распространенный случай. Тогда мы смогли бы ограничиться только изменениями класса `Glyph` и тех его подклассов, которые отклоняются от нормы.

Несмотря на то что реализация по умолчанию сокращает объем изменений, принципиальная проблема остается: интерфейс класса `Glyph` необходимо расширять при добавлении каждого нового вида анализа. Со временем такие операции начнут скрывать смысл этого интерфейса. Будет трудно понять, что основная цель глифа — определить и структурировать объекты, имеющие внешнее представление и форму; интерфейс потеряется за посторонним шумом.

ИНКАПСУЛЯЦИЯ АНАЛИЗА

Судя по всему, стоит инкапсулировать анализ в отдельный объект, как мы уже много раз делали прежде. Можно было бы поместить механизм конкретного вида анализа в его собственный класс, а экземпляр этого класса использовать совместно с подходящим итератором. Тогда итератор «переносил» бы этот экземпляр от одного глифа к другому, а объект выполнял бы свой анализ для каждого элемента. По мере продвижения обхода анализатор накапливал бы определенную информацию (в данном случае символы).



Принципиальный вопрос при таком подходе — как объект-анализатор различает виды глифов, не прибегая к проверке или приведениям типов? Мы не хотим включать в класс `SpellingChecker` псевдокод следующего вида:

```
void SpellingChecker::Check (Glyph* glyph) {
    Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
        // проанализировать символ

    } else if (r = dynamic_cast<Row*>(glyph)) {
        // подготовиться к анализу потомков r

    } else if (i = dynamic_cast<Image*>(glyph)) {
        // ничего не делать
    }
}
```

Такой код получается довольно уродливым. Он опирается на специфические возможности вроде безопасных по отношению к типам приведений. Его трудно расширять. Нужно не забыть изменить тело данной функции после любого изменения иерархии класса `Glyph`. В общем это как раз такой код, для избавления от которого создавались объектно-ориентированные языки.

Как уйти от данного подхода методом «грубой силы»? Посмотрим, что произойдет, если мы добавим в класс `Glyph` такую абстрактную операцию:

```
void CheckMe(SpellingChecker&)
```

Определим операцию `CheckMe` в каждом подклассе класса `Glyph` следующим образом:

```
void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```

где `GlyphSubclass` заменяется именем подкласса глифа. Заметим, что при вызове `CheckMe` конкретный подкласс класса `Glyph` известен, ведь мы же выполняем одну из его операций. В свою очередь, в интерфейсе класса `SpellingChecker` есть операция типа `CheckGlyphSubclass` для каждого подкласса класса `Glyph`[1]:

---

[1] Можно было бы воспользоваться перегрузкой функций, чтобы присвоить этим функциям одинаковые имена, поскольку их можно различить по типам параметров. Здесь мы дали им разные имена, чтобы было видно, что это все-таки разные функции, особенно при их вызове.

```
class SpellingChecker {
public:
    SpellingChecker();
    virtual void CheckCharacter(Character*);
    virtual void CheckRow(Row*);
    virtual void CheckImage(Image*);

    // ...и так далее

    List<char*>& GetMisspellings();

protected:
    virtual bool IsMisspelled(const char*);

private:
    char _currentWord[MAX_WORD_SIZE];
    List<char*> _misspellings;
};
```

Операция проверки в классе `SpellingChecker` для глифов типа `Character` могла бы выглядеть так:

```
void SpellingChecker::CheckCharacter (Character* c) {
    const char ch = c->GetCharCode();

    if (isalpha(ch)) {
        // присоединить алфавитный символ к _currentWord
    } else {
        // обнаружен символ, не являющийся алфавитным

        if (IsMisspelled(_currentWord)) {
            // добавить _currentWord к _misspellings
            _misspellings.Append(strdup(_currentWord));
        }

        _currentWord[0] = '\0';
            // сбросить _currentWord для проверки следующего слова
    }
}
```

Обратите внимание: мы определили специальную операцию `GetCharCode` только для класса `Character`. Объект проверки правописания может работать со специфическими для подклассов операциями, не прибегая к проверке или приведению типов, а это позволяет нам трактовать некоторые объекты специальным образом.

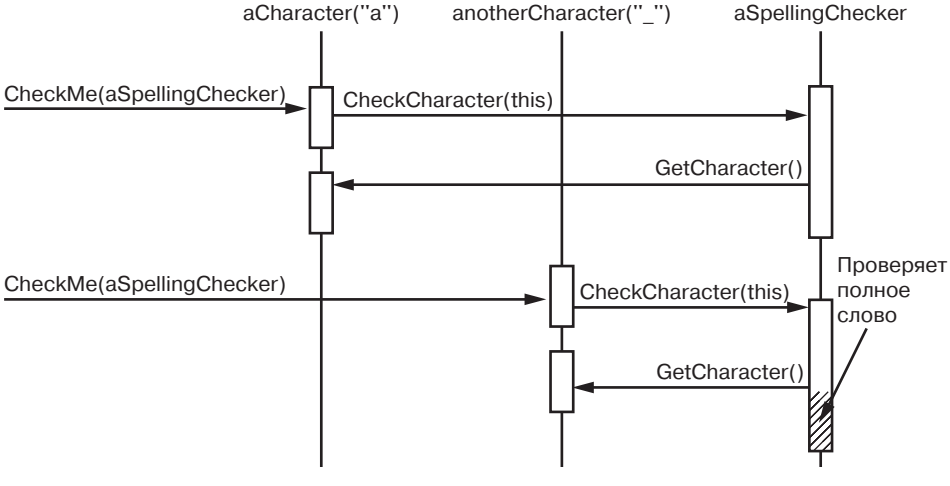
Объект класса `CheckCharacter` накапливает буквы в буфере `_currentWord`. Когда встречается не буква, например символ подчеркивания, этот объект вызывает операцию `IsMisspelled` для проверки орфографии слова, находя-

щегося в `_currentWord`[1]. Если слово написано неправильно, то `CheckCharacter` добавляет его в список слов с ошибками. Затем буфер `_currentWord` очищается для приема следующего слова. По завершении обхода можно добраться до списка слов с ошибками с помощью операции `GetMisspellings`.

Теперь логично обойти всю структуру глифов, вызывая `CheckMe` для каждого глифа и передавая ей объект проверки правописания в качестве аргумента. Тем самым текущий глиф для `SpellingChecker` идентифицируется и приказывает модулю проверки выполнить следующий шаг в проверке:

```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);

for (i.First(); !i.IsDone(); i.Next()) {
    g = i.CurrentItem();
    g->CheckMe(spellingChecker);
}
```

На следующей схеме показано, как взаимодействуют глифы типа `Character` и объект `SpellingChecker`.



[1] Функция `IsMisspelled` реализует алгоритм проверки орфографии, подробности которого здесь не приводятся, поскольку мы сделали его независимым от дизайна Lexi. Мы можем поддержать разные алгоритмы, порождая подклассы класса `SpellingChecker`, или применить для этой цели паттерн стратегия (как для форматирования в разделе 2.3).

Этот подход работает при поиске орфографических ошибок, но как он может помочь в поддержке нескольких видов анализа? Похоже, что придется добавлять операцию вроде `CheckMe(SpellingChecker&)` в класс `Glyph` и его подклассы всякий раз, когда добавляется новый вид анализа. Так оно и есть, если мы настаиваем на *независимом* классе для каждого вида анализа. Но почему бы не придать *всем* видам анализа одинаковый интерфейс? Это позволит нам использовать их полиморфно. И тогда мы сможем заменить специфические для конкретного вида анализа операции вроде `CheckMe(SpellingChecker&)` одной инвариантной операцией, принимающей более общий параметр.

КЛАСС VISITOR И ЕГО ПОДКЛАССЫ

Мы будем использовать термин «посетитель» для обозначения класса объектов, «посещающих» другие объекты во время обхода, дабы сделать то, что необходимо в данном контексте[1]. Тогда мы можем определить класс `Visitor`, описывающий абстрактный интерфейс для посещения глифов в структуре:

```
class Visitor {
public:
    virtual void VisitCharacter(Character*) { }
    virtual void VisitRow(Row*) { }
    virtual void VisitImage(Image*) { }

    // ...И так далее
    };
```

Конкретные подклассы `Visitor` выполняют разные виды анализа. Например, можно определить подкласс `SpellingCheckingVisitor` для проверки правописания и подкласс `HyphenationVisitor` для расстановки переносов. При этом `SpellingCheckingVisitor` был бы реализован точно так же, как мы реализовали класс `SpellingChecker` выше, только имена операций отражали бы более общий интерфейс класса `Visitor`. Так, операция `CheckCharacter` называлась бы `VisitCharacter`.

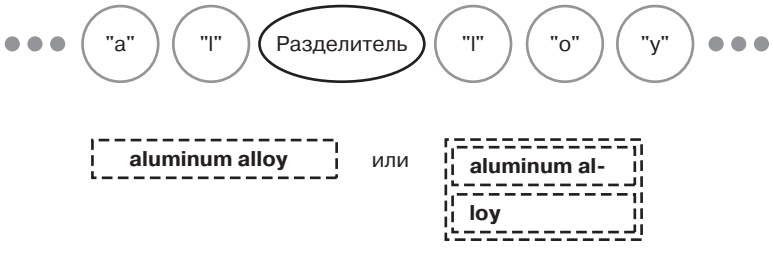
Поскольку имя `CheckMe` не подходит для посетителей, которые ничего не проверяют, мы используем имя `Accept`. Аргумент этой операции тоже придется изменить на `Visitor&`, чтобы отразить тот факт, что может приниматься любой посетитель. Теперь для добавления нового вида анализа нужно лишь определить новый подкласс класса `Visitor`, а трогать классы глифов вовсе

[1] «Посетить» — всего лишь чуть более общий термин, чем «проанализировать». Он просто предвосхищает ту терминологию, которая будет использоваться при обсуждении следующего паттерна.

не обязательно. Таким образом добавление всего одной операции в класс `Glyph` и его подклассы позволяет поддерживать в будущем все возможные виды анализа.

О том, как работает проверка правописания, говорилось выше. Такой же подход будет применен для накопления текста в подклассе `HyphenationVisitor`. Но после того как операция `VisitCharacter` из подкласса `HyphenationVisitor` закончила распознавание целого слова, она ведет себя по-другому. Вместо проверки орфографии применяется алгоритм расстановки переносов, чтобы определить, в каких местах можно перенести слово на другую строку (если это вообще возможно). Затем для каждой из найденных точек в структуру вставляется разделяющий (discretionary) глиф. Разделяющие глифы являются экземплярами подкласса `Glyph` — класса `Discretionary`.

Разделяющий глиф может выглядеть по-разному в зависимости от того, является он последним символом в строке или нет. Если это последний символ, глиф выглядит как дефис, в противном случае он не отображается вообще. Разделяющий глиф проверяет своего родителя (объект `Row`), чтобы узнать, является ли он последним потомком; он делает это всякий раз, когда от него требуют отобразить себя или вычислить свои размеры. Стратегия форматирования поступает с разделяющими глифами точно так же, как с пропусками, в результате чего они становятся «кандидатами» на завершающий символ строки. На схеме ниже показано, как может выглядеть встроенный разделитель.



ПАТТЕРН VISITOR (ПОСЕТИТЕЛЬ)

Вышеописанная процедура — пример применения паттерна посетитель (379). Его главными составляющими являются класс `Visitor` и его подклассы. Паттерн посетитель абстрагирует метод, позволяющий иметь произвольное число видов анализа структур глифов без изменения самих классов глифов. Еще одна полезная особенность посетителей состоит в том, что их можно применять не только к таким агрегатам, как наши структуры

глифов, но и к *любым* структурам, состоящим из объектов. В эту категорию входят множества, списки и даже направленные ациклические графы. Более того, классы, которые обходит посетитель, необязательно должны быть связаны друг с другом через общий родительский класс. А это значит, что посетители могут охватывать разные иерархии классов.

Важный вопрос, который надо задать себе перед применением паттерна посетитель, звучит так: «Какие иерархии классов наиболее часто будут изменяться?» Этот паттерн особенно хорошо подходит для выполнения действий с объектами, входящими в стабильную структуру классов. Добавление нового вида посетителя не требует изменять структуру классов, что особенно важно, когда эта структура велика. Но каждый раз, когда в структуру добавляется новый подкласс, вам придется обновить все интерфейсы посетителя и добавить операцию `Visit`... для этого подкласса. В нашем примере это означает, что добавление подкласса `Foo` класса `Glyph` потребует изменить класс `Visitor` и все его подклассы, чтобы добавить операцию `VisitFoo`. Однако при наших ограничениях гораздо более вероятно добавление к Lexi нового вида анализа, а не нового вида глифов. Поэтому для наших целей паттерн посетитель вполне подходит.

2.9. РЕЗЮМЕ

При проектировании Lexi мы применили восемь различных паттернов:

- компоновщик (196) для представления физической структуры документа;
- стратегия (362) для возможности использования различных алгоритмов форматирования;
- декоратор (209) для оформления пользовательского интерфейса;
- абстрактная фабрика (113) для поддержки нескольких стандартов оформления;
- мост (184) для поддержки нескольких оконных систем;
- команда (275) для реализации отмены и повтора операций пользователя;
- итератор (302) для обхода структур объектов;
- посетитель (379) для поддержки неизвестного заранее числа видов анализа без усложнения реализации структуры документа.

Ни одно из этих проектных решений не ограничено документо-ориентированными редакторами вроде Lexi. На самом деле в большинстве нетривиальных приложений есть возможность воспользоваться многими из этих паттернов, быть может, для других целей. В приложении для финансового анализа паттерн компоновщик можно было бы применить для определения инвестиционных портфелей, разбитых на субпортфели и счета разных видов. Компилятор мог бы использовать паттерн стратегия, чтобы поддержать реализацию разных схем распределения машинных регистров для целевых компьютеров с различной архитектурой. Приложения с графическим интерфейсом пользователя вполне могли бы применить паттерны декоратор и команда точно так же, как это сделали мы.

Хотя мы и рассмотрели несколько крупных проблем проектирования Lexi, осталось гораздо больше таких, которых мы не касались. Но ведь и в книге описаны не только рассмотренные восемь паттернов. Поэтому, изучая остальные паттерны, подумайте о том, как вы могли бы применить их к Lexi. А еще лучше — подумайте об их использовании в своих собственных проектах!

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

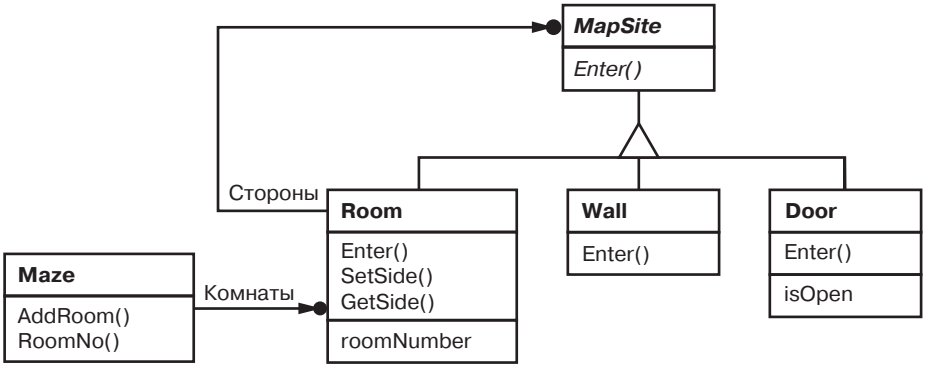
использовать как прототип (146), так и абстрактную фабрику (113). В других ситуациях порождающие паттерны дополняют друг друга. Так, применяя паттерн строитель (124), можно использовать другие паттерны для решения вопроса о том, какие компоненты нужно строить, а прототип (146) может использовать одиночку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

порождающих паттернов можно различными способами избавиться от явных упоминаний конкретных классов из кода, создающего их экземпляры:

- если `CreateMaze` вызывает виртуальные функции вместо конструкторов для создания комнат, стен и дверей, то классы, экземпляры которых создаются, можно подменить, создав подкласс `MazeGame` и переопределив в нем виртуальные функции. Такой подход применяется в паттерне фабричный метод (135);
- когда функции `CreateMaze` в параметре передается объект, используемый для создания комнат, стен и дверей, то их классы можно изменить, передав другой параметр. Это пример паттерна абстрактная фабрика (113);
- если функции `CreateMaze` передается объект, способный целиком создать новый лабиринт с помощью своих операций для добавления комнат, дверей и стен, можно воспользоваться наследованием для изменения частей лабиринта или способа его построения. Такой подход применяется в паттерне строитель (124);
- если `CreateMaze` параметризована прототипами комнаты, двери и стены, которые она затем копирует и добавляет к лабиринту, то состав лабиринта можно варьировать, заменяя одни объекты-прототипы другими. Это паттерн прототип (146).

Последний из порождающих паттернов, одиночка (157), может гарантировать существование единственного лабиринта в игре и свободный доступ к нему со стороны всех игровых объектов, не прибегая к глобальным переменным или функциям. Паттерн одиночка также позволяет легко расширить или заменить лабиринт, не трогая существующий код.

ПАТТЕРН ABSTRACT FACTORY (АБСТРАКТНАЯ ФАБРИКА)

Название и классификация паттерна

Абстрактная фабрика — паттерн, порождающий объекты.

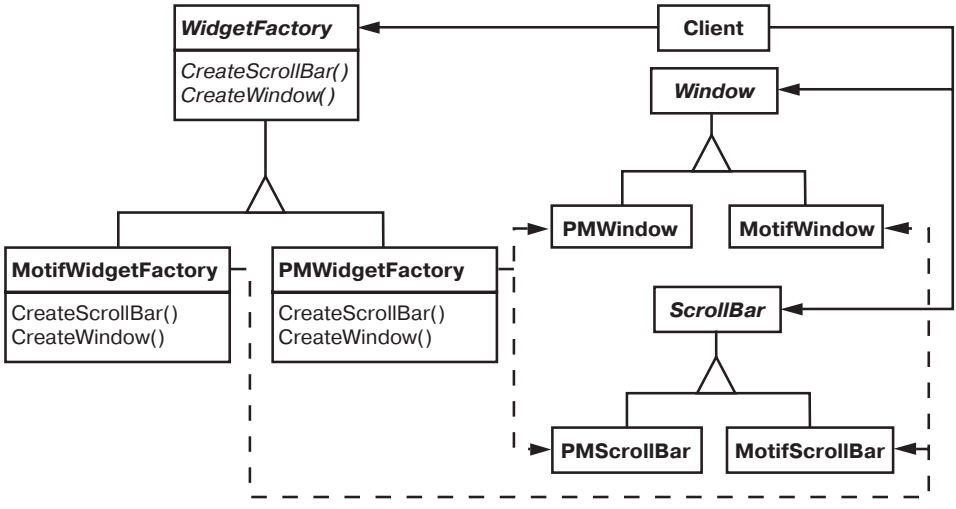
Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Другие названия

`Kit` (инструментарий).

Мотивация



Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты оформления, например Motif и Presentation Manager. Оформление определяет визуальное представление и поведение элементов пользовательского интерфейса («виджетов») — полос прокрутки, окон и кнопок. Чтобы приложение можно было перенести на другой стандарт, в нем не должен быть жестко закодировано оформление виджетов. Если создание экземпляров классов для конкретного оформления разбросано по всему приложению, то изменить оформление впоследствии будет нелегко.

Для решения этой проблемы можно определить абстрактный класс `WidgetFactory`, в котором объявлен интерфейс для создания всех основных видов виджетов. Есть также абстрактные классы для каждого отдельного вида и конкретные подклассы, реализующие виджеты с определенным оформлением. В интерфейсе `WidgetFactory` имеется операция, возвращающая новый объект-виджет для каждого абстрактного класса виджетов. Клиенты вызывают эти операции для получения экземпляров виджетов, но при этом ничего не знают о том, какие именно классы используются. Таким образом, клиенты остаются независимыми от выбранного стандарта оформления.

Для каждого стандарта оформления существует определенный подкласс `WidgetFactory`. Каждый такой подкласс реализует операции, необходимые для создания соответствующего стандарту виджета. Например, операция `CreateScrollBar` в классе `MotifWidgetFactory` создает экземпляр и возвращает полосу прокрутки в стандарте Motif, тогда как соответствующая опе-

рация в классе `PMWidgetFactory` возвращает полосу прокрутки в стандарте Presentation Manager. Клиенты создают виджеты, пользуясь исключительно интерфейсом `WidgetFactory`, и им ничего не известно о классах, реализующих виджеты для конкретного стандарта. Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

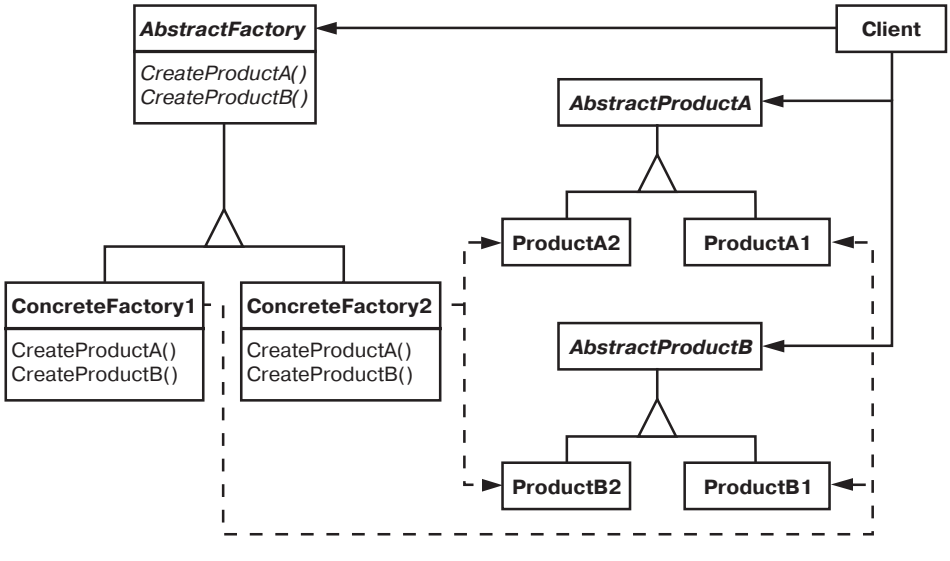
Класс `WidgetFactory` также устанавливает зависимости между конкретными классами виджетов. Полоса прокрутки для Motif должна использоваться с кнопкой и текстовым полем Motif, и это ограничение поддерживается автоматически как следствие использования класса `MotifWidgetFactory`.

Применимость

Основные условия для применения паттерна абстрактная фабрика:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- система должна настраиваться одним из семейств объектов;
- входящие в семейство взаимосвязанные объекты спроектированы для совместной работы, и вы должны обеспечить выполнение этого ограничения;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура



Участники

- **AbstractFactory** (`WidgetFactory`) — абстрактная фабрика:
  - объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- **ConcreteFactory** (`MotifWidgetFactory`, `PMWidgetFactory`) — конкретная фабрика:
  - реализует операции, создающие конкретные объекты-продукты;
- **AbstractProduct** (`Window`, `ScrollBar`) — абстрактный продукт:
  - объявляет интерфейс для типа объекта-продукта;
- **ConcreteProduct** (`MotifWindow`, `MotifScrollBar`) — конкретный продукт:
  - определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
  - реализует интерфейс `AbstractProduct`;
- **Client** — клиент:
  - пользуется исключительно интерфейсами, которые объявлены в классах `AbstractFactory` и `AbstractProduct`.

Отношения

- Обычно во время выполнения создается единственный экземпляр класса `ConcreteFactory`. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;
- `AbstractFactory` передоверяет создание объектов-продуктов своему подклассу `ConcreteFactory`.

Результаты

Паттерн абстрактная фабрика:

- *изолирует конкретные классы*. Паттерн помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от подробностей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;

- *упрощает замену семейств продуктов.* Класс конкретной фабрики появляется в приложении только один раз: при создании экземпляра. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере для переключения пользовательского интерфейса с виджетов Motif на виджеты Presentation Manager достаточно переключиться на продукты соответствующей фабрики и заново создать интерфейс;
- *гарантирует сочетаемость продуктов.* Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс `AbstractFactory` позволяет легко соблюсти это ограничение;
- *не упрощает задачу поддержки нового вида продуктов.* Расширение абстрактной фабрики для изготовления новых видов продуктов — непростая задача. Дело в том, что интерфейс `AbstractFactory` фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс `AbstractFactory` и все его подклассы. Одно из возможных решений этой проблемы рассматривается в разделе «Реализация».

Реализация

Некоторые полезные приемы реализации паттерна абстрактная фабрика:

- *фабрики как объекты, существующие в единственном экземпляре.* Как правило, приложению нужен только один экземпляр класса `ConcreteFactory` на каждое семейство продуктов. Поэтому для реализации лучше всего применить паттерн одиночка (157);
- *создание продуктов.* Класс `AbstractFactory` объявляет только *интерфейс* для создания продуктов. Фактическое их создание — дело подклассов `ConcreteProduct`. Чаще всего для этой цели определяется фабричный метод для каждого продукта (см. паттерн фабричный метод (135)). Конкретная фабрика определяет свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются.

  Если семейств продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна прототип (146). В этом случае

она инициализируется экземпляром-прототипом каждого продукта в семействе и создает новый продукт путем клонирования этого прототипа. Подход на основе прототипов устраняет необходимость создавать новый класс конкретной фабрики для каждого нового семейства продуктов.

Вот как можно реализовать фабрику на основе прототипов в языке Smalltalk. Конкретная фабрика хранит клонируемые прототипы в словаре под названием `partCatalog`. Метод `make:` извлекает прототип и клонирует его:

```
make: partName
    ^ (partCatalog at: partName) copy
```

У конкретной фабрики есть метод для добавления деталей в каталог:

```
addPart: partTemplate named: partName
    partCatalog at: partName put: partTemplate
```

Прототипы добавляются к фабрике путем пометки их символом #:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

Разновидность метода на базе прототипов возможна в языках, в которых классы являются полноценными объектами (например, Smalltalk и Objective C). В таких языках класс можно представлять себе как вырожденный случай фабрики, умеющей создавать только один вид продуктов. Можно хранить *классы* внутри конкретной фабрики, которая создает разные конкретные продукты в переменных. Это очень похоже на прототипы. Такие классы создают новые экземпляры от имени конкретной фабрики. Новая фабрика инициализируется экземпляром конкретной фабрики с *классами* продуктов, а не путем порождения подкласса. Подобный подход задействует некоторые специфические свойства языка, тогда как «чистый» подход, основанный на прототипах, от языка не зависит.

Как и для только что рассмотренной фабрики на базе прототипов в Smalltalk, в версии на основе классов будет единственная переменная экземпляра `partCatalog`, представляющая собой словарь, ключом которого является название детали. Но вместо хранения клонируемых прототипов `partCatalog` хранит классы продуктов. Метод `make:` выглядит теперь следующим образом:

```
make: partName
    ^ (partCatalog at: partName) new
```

- *определение расширяемых фабрик.* Класс `AbstractFactory` обычно определяет разные операции для всех видов изготавливаемых продуктов. Виды продуктов кодируются в сигнатуре операции. Для добавления нового вида продуктов нужно изменить интерфейс класса `AbstractFactory` и всех зависящих от него классов.

  Более гибкий, но менее безопасный способ — добавить параметр к операциям, создающим объекты. Данный параметр определяет вид создаваемого объекта. Это может быть идентификатор класса, целое число, строка или что-то еще, однозначно описывающее вид продукта. При таком подходе классу `AbstractFactory` нужна только одна операция `Make` с параметром, задающим тип создаваемого объекта. Данный прием применялся в обсуждавшихся выше абстрактных фабриках на основе прототипов и классов.

  Такой вариант проще использовать в динамически типизированных языках вроде Smalltalk, нежели в статически типизированных, каким является C++. Воспользоваться им в C++ можно только в том случае, если у всех объектов имеется общий абстрактный базовый класс или если объекты-продукты могут быть безопасно приведены к корректному типу клиентом, который их запросил. В разделе «Реализация» из описания паттерна фабричный метод (135) показано, как реализовать такие параметризованные операции в C++.

Но даже если приведение типов не нужно, остается принципиальная проблема: все продукты возвращаются клиенту *одним и тем же* абстрактным интерфейсом с уже определенным типом возвращаемого значения. Клиент не может ни различить классы продуктов, ни сделать какие-нибудь предположения о них. Если клиенту нужно выполнить операцию, зависящую от подкласса, то она будет недоступна через абстрактный интерфейс. Хотя клиент мог бы выполнить понижающее приведение типа (например, с помощью оператора `dynamic_cast` в C++), это небезопасно и необязательно заканчивается успешно. Здесь мы имеем классический пример компромисса между высокой степенью гибкости и расширяемостью интерфейса.

Пример кода

Паттерн абстрактная фабрика мы применим к построению обсуждавшихся в начале этой главы лабиринтов.

Класс `MazeFactory` может создавать компоненты лабиринтов. Он строит комнаты, стены и двери между комнатами. Например, им можно воспользоваться из программы, которая читает план лабиринта из файла, а затем

создает его, или из приложения, строящего случайный лабиринт. Программы построения лабиринта принимают `MazeFactory` в качестве аргумента, так что программист может сам указать классы комнат, стен и дверей:

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

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Напомним, что функция `CreateMaze` строит небольшой лабиринт, состоящий всего из двух комнат, соединенных одной дверью. В ней жестко «зашиты» имена классов, поэтому воспользоваться функцией для создания лабиринтов с другими компонентами проблематично.

Следующая версия `CreateMaze` избавлена от подобного недостатка, поскольку она получает `MazeFactory` в параметре:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

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

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

Чтобы создать фабрику `EnchantedMazeFactory` для производства волшебных лабиринтов, следует породить подкласс от `MazeFactory`. В этом подклассе замещены различные функции класса, так что он возвращает другие подклассы классов `Room`, `Wall` и т. д.:

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n)  const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2)  const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

А теперь предположим, что мы хотим построить для некоторой игры лабиринт, в комнате которого может быть заложена бомба. Если бомба взрывается, то она разрушает стены (а то и что-нибудь еще). Тогда можно породить от класса `Room` подкласс, отслеживающий, есть ли в комнате бомба и взорвалась ли она. Также понадобится подкласс класса `Wall` для хранения информации о том, были ли повреждены стены. Назовем эти классы соответственно `RoomWithABomb` и `BombedWall`.

И наконец, мы определим класс `BombedMazeFactory`, являющийся подклассом `BombedMazeFactory`, который создает стены класса `BombedWall` и комнаты класса `RoomWithABomb`. В `BombedMazeFactory` надо переопределить всего две функции:

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

Чтобы построить простой лабиринт, в котором могут быть спрятаны бомбы, просто вызовем функцию `CreateMaze`, передав ей в параметре `BombedMazeFactory`:

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

С таким же успехом `CreateMaze` можно передать в параметре `EnchantedMazeFactory` для построения волшебного лабиринта.

Отметим, что `MazeFactory` — всего лишь набор фабричных методов. Это самый распространенный способ реализации паттерна абстрактная фабрика. Еще заметим, что MazeFactory — не абстрактный класс, то есть он работает и как `AbstractFactory`, и как `ConcreteFactory`. Это еще одна типичная реализация для простых применений паттерна абстрактная фабрика. Поскольку `MazeFactory` — конкретный класс, состоящий только из фабричных методов, легко получить новую фабрику `MazeFactory`, породив подкласс и заместив в нем необходимые операции.

В функции `CreateMaze` используется операция `SetSide` для описания сторон комнат. Если она создает комнаты с помощью фабрики `BombedMazeFactory`, то лабиринт будет составлен из объектов класса `RoomWithABomb`, стороны которых описываются объектами класса `BombedWall`. Если классу `RoomWithABomb` потребуется обратиться к членам `BombedWall`, не имеющим аналога в его предках, то ссылку на объекты-стены придется преобразовать от типа `Wall*` к типу `BombedWall*`. Такое понижающее приведение безопасно при условии, что аргумент действительно принадлежит классу `BombedWall*`, а это заведомо так, если стены создаются исключительно фабрикой `BombedMazeFactory`.

В динамически типизированных языках вроде Smalltalk приведение, разумеется, не нужно, но будет выдано сообщение об ошибке во время выполнения, если объект класса `Wall` встретится вместо ожидаемого объекта *подкласса* класса `Wall`. Использование абстрактной фабрики для создания стен предотвращает подобные ошибки, гарантируя, что могут быть созданы лишь стены определенных типов.

Рассмотрим версию `MazeFactory` на языке Smalltalk, в которой есть единственная операция `make`, получающая вид изготавливаемого объекта в параметре. Конкретная фабрика при этом будет хранить классы изготавливаемых объектов.

Для начала напишем на Smalltalk эквивалент `CreateMaze`:

```
createMaze: aFactory
    | room1 room2 aDoor |
    room1 := (aFactory make: #room) number: 1.
    room2 := (aFactory make: #room) number: 2.
    aDoor := (aFactory make: #door) from: room1 to: room2.
    room1 atSide: #north put: (aFactory make: #wall).
    room1 atSide: #east put: aDoor.
    room1 atSide: #south put: (aFactory make: #wall).
```

```
    room1 atSide: #west put: (aFactory make: #wall).
    room2 atSide: #north put: (aFactory make: #wall).
    room2 atSide: #east put: (aFactory make: #wall).
    room2 atSide: #south put: (aFactory make: #wall).
    room2 atSide: #west put: aDoor.
    ^ Maze new addRoom: room1; addRoom: room2; yourself
```

В разделе «Реализация» мы уже говорили о том, что классу `MazeFactory` нужна всего одна переменная экземпляра `partCatalog`, предоставляющая словарь, в котором ключом служит класс компонента. Напомним еще раз реализацию метода `make`:

```
make: partName
    ^ (partCatalog at: partName) new
```

Теперь мы можем создать фабрику `MazeFactory` и воспользоваться ей для реализации `createMaze`. Для создания фабрики будет использоваться метод `createMazeFactory` класса `MazeGame`:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

`BombedMazeFactory` и `EnchantedMazeFactory` создаются путем ассоциирования других классов с ключами. Например, `EnchantedMazeFactory` можно создать следующим образом:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

Известные применения

В библиотеке InterViews [Lin92] для обозначения классов абстрактных фабрик используется суффикс «Kit». Для изготовления объектов пользовательского интерфейса с заданным внешним обликом в ней определены абстрактные фабрики `WidgetKit` и `DialogKit`. В InterViews также включен класс `LayoutKit`, который генерирует разные объекты композиции в зависимости от того, какая требуется стратегия размещения. Например, разме-

щение, которое концептуально можно было бы назвать «горизонтальным», может потребовать разных объектов в зависимости от ориентации документа (книжной или альбомной).

В библиотеке ET++ [WGM88] паттерн абстрактная фабрика применяется для достижения переносимости между разными оконными системами (например, X Window и SunView). Абстрактный базовый класс `WindowSystem` определяет интерфейс для создания объектов, которые представляют ресурсы оконной системы (`MakeWindow`, `MakeFont`, `MakeColor` и т. п.). Его конкретные подклассы реализуют эти интерфейсы для той или иной оконной системы. Во время выполнения ET++ создает экземпляр конкретного подкласса `WindowSystem`, который уже и порождает объекты, соответствующие ресурсам данной оконной системы.

Родственные паттерны

Классы `AbstractFactory` часто реализуются фабричными методами (см. паттерн фабричный метод (135)), но могут быть реализованы и с помощью паттерна прототип (146).

Конкретная фабрика часто описывается паттерном одиночка (157).

ПАТТЕРН BUILDER (СТРОИТЕЛЬ)

Название и классификация паттерна

Строитель — паттерн, порождающий объекты.

Назначение

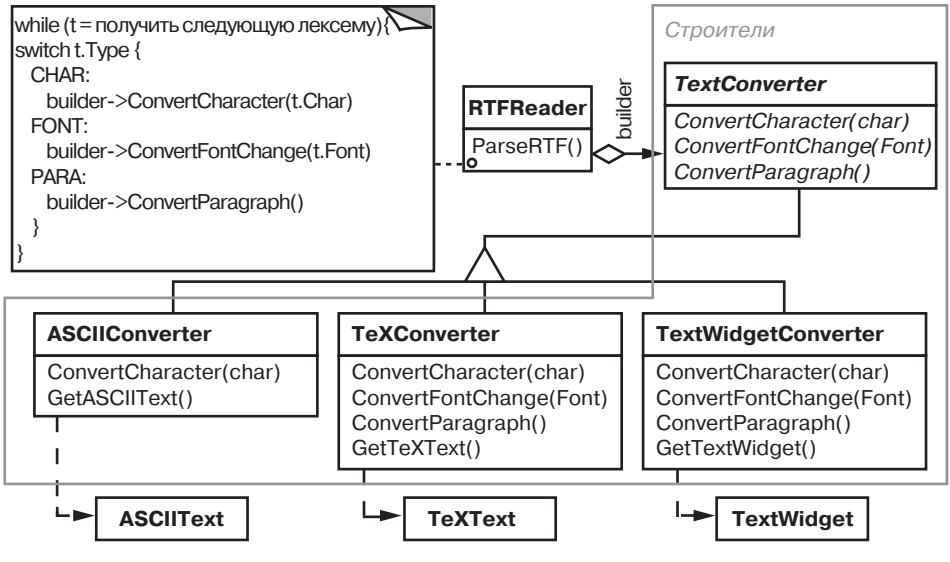
Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Мотивация

Программа чтения документов в формате RTF (Rich Text Format) должна также «уметь» преобразовывать его во многие другие форматы — например, в простой ASCII текст или в представление, которое можно отобразить в виджете для ввода текста. Тем не менее, число возможных преобразований заранее неизвестно, поэтому новые преобразования должны легко добавляться без изменения программы.

Проблема решается настройкой класса `RTFReader` с помощью объекта `TextConverter`, который мог бы преобразовывать RTF в другой текстовый формат. При разборе документа в формате RTF класс `RTFReader` вызывает `TextConverter` для выполнения преобразования. Всякий раз, как `RTFReader` распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту `TextConverter` посылается запрос. Объекты `TextConverter` отвечают как за преобразование данных, так и за представление лексемы в конкретном формате.

Подклассы `TextConverter` специализируются на различных преобразованиях и форматах. Например, `ASCIIConverter` игнорирует запросы на преобразование чего бы то ни было, кроме простого текста. С другой стороны, `TeXConverter` будет реализовывать операции для всех запросов на получение представления в формате редактора TeX, собирая по ходу необходимую информацию о стилях. А `TextWidgetConverter` будет строить сложный объект пользовательского интерфейса, который позволит пользователю просматривать и редактировать текст.



Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа.

В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется *строителем*, а загрузчик — *распорядителем*. В применении к рассмотренному примеру строитель отделяет алгоритм интерпре-

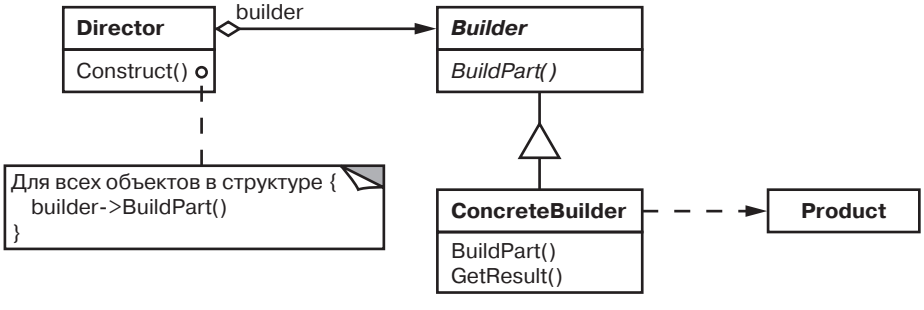
тации формата текста (то есть парсер RTF-документов) от того, как создается и представляется документ в преобразованном формате. Это позволяет повторно использовать алгоритм разбора, реализованный в `RTFReader`, для создания разных текстовых представлений RTF-документов; достаточно передать в `RTFReader` различные подклассы класса `TextConverter`.

Применимость

Основные условия для применения паттерна строитель:

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Структура



Участники

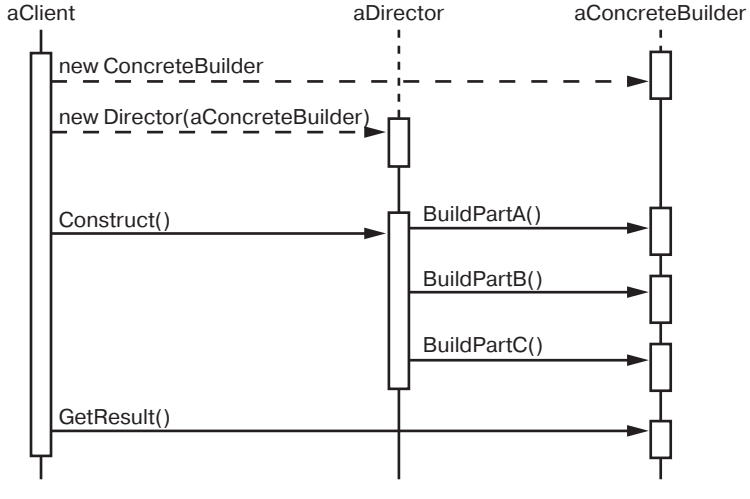
- **Builder** (`TextConverter`) — строитель:
  - задает абстрактный интерфейс для создания частей объекта Product;
- **ConcreteBuilder** (`ASCIIConverter`, `TeXConverter`, `TextWidgetConverter`) — конкретный строитель:
  - конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;
  - определяет создаваемое представление и следит за ним;
  - предоставляет интерфейс для доступа к продукту (например, `GetASCIIText`, `GetTextWidget`);
- **Director** (`RTFReader`) — распорядитель:
  - конструирует объект, пользуясь интерфейсом Builder;

- **Product** (`ASCIIText`, `TeXText`, `TextWidget`) — продукт:
  - представляет сложный конструируемый объект. `ConcreteBuilder` строит внутреннее представление продукта и определяет процесс его сборки;
  - включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

Отношения

- клиент создает объект-распорядитель `Director` и настраивает его нужным объектом-строителем `Builder`;
- распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;
- строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;
- клиент забирает продукт у строителя.

Следующая схема взаимодействий иллюстрирует взаимоотношения строителя и распорядителя с клиентом.



Результаты

Паттерн строитель:

- *позволяет изменять внутреннее представление продукта.* Объект `Builder` предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представле-

ние и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя;

- *изолирует код, реализующий конструирование и представление*. Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта; эти классы не входят в интерфейс строителя.

  Каждый конкретный строитель `ConcreteBuilder` содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей. В примере с RTF-документом мы могли бы определить загрузчик для формата, отличного от RTF (скажем, `SGMLReader`), и воспользоваться теми же классами `TextConverter` для генерирования представлений SGML-документов в виде ASCII-текста, TeX-текста или текстового виджета;

- *предоставляет более точный контроль над процессом конструирования*. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

Реализация

Обычно существует абстрактный класс `Builder`, в котором определены операции для каждого компонента, который может потребовать создать распорядитель. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя `ConcreteBuilder` они замещены для тех компонентов, в создании которых он принимает участие.

Также существуют и другие аспекты реализации, заслуживающие внимания:

- *интерфейс сборки и конструирования*. Строители конструируют свои продукты шаг за шагом, поэтому интерфейс класса `Builder` должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя.

Ключевой аспект проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто присоединяются к продукту. В примере с RTF-документами строитель преобразует и добавляет очередную лексему к уже конвертированному тексту.

Но иногда может потребоваться доступ к отдельным частям сконструированного к данному моменту продукта. В примере с лабиринтом, который будет описан в разделе «Пример кода», интерфейс класса `MazeBuilder` позволяет добавлять дверь между уже существующими комнатами. Другим примером являются древовидные структуры — скажем, деревья синтаксического разбора, которые строятся снизу вверх. В этом случае строитель возвращает узлы-потомки распорядителю, который затем передает их обратно строителю, чтобы тот мог построить родительские узлы;

- *почему нет абстрактного класса для продуктов*? В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. В примере с RTF-документами трудно представить себе общий интерфейс у объектов `ASCIIText` и `TextWidget`, да он и не нужен. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса `Builder` используется и как нужно обращаться с произведенными продуктами;
- *пустые методы класса Builder по умолчанию*. В C++ методы строителя намеренно не объявлены чисто виртуальными функциями. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, которые представляют для него интерес.

Пример кода

Определим вариант функции `CreateMaze`, которая получает в аргументе строителя, принадлежащего классу `MazeBuilder`.

Класс `MazeBuilder` определяет следующий интерфейс для построения лабиринтов:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
```

```
     virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Этот интерфейс позволяет создавать три сущности: лабиринт, комнату с конкретным номером, двери между пронумерованными комнатами. Операция `GetMaze` возвращает лабиринт клиенту. В подклассах `MazeBuilder` данная операция переопределяется для возвращения реально созданного лабиринта.

Все операции построения лабиринта в классе `MazeBuilder` по умолчанию ничего не делают. Но они не объявлены чисто виртуальными, чтобы в производных классах можно было замещать лишь часть методов.

Имея интерфейс `MazeBuilder`, можно изменить функцию `CreateMaze`, чтобы она получала строителя в параметре:

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Сравните эту версию `CreateMaze` с первоначальной. Обратите внимание, как строитель скрывает внутреннее представление лабиринта, то есть классы комнат, дверей и стен, и как эти части собираются вместе для завершения построения лабиринта. Кто-то, может, и догадается, что для представления комнат и дверей есть особые классы, но ничто не указывает на существование такого класса для стен. За счет этого становится проще модифицировать способ представления лабиринта, поскольку ни один из клиентов `MazeBuilder` изменять не надо.

Как и другие порождающие паттерны, строитель инкапсулирует способ создания объектов — в данном случае с помощью интерфейса, определенного классом `MazeBuilder`. Это означает, что `MazeBuilder` может повторно использоваться для построения лабиринтов разных видов. В качестве примера возьмем функцию `CreateComplexMaze`:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
```

```
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Обратите внимание: `MazeBuilder` не создает лабиринты самостоятельно, его основная цель — просто определить интерфейс для создания лабиринтов. Пустые реализации в этом интерфейсе определены только для удобства. Реальную работу выполняют подклассы `MazeBuilder`.

Подкласс `StandardMazeBuilder` содержит реализацию построения простых лабиринтов. Создаваемый лабиринт хранится в переменной `_currentMaze`:

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

`CommonWall` (общая стена) — это вспомогательная операция, которая определяет направление общей стены между двумя комнатами.

Конструктор `StandardMazeBuilder` просто инициализирует `_currentMaze`:

```
StandardMazeBuilder::StandardMazeBuilder () {
   _currentMaze = 0;
}
```

`BuildMaze` создает экземпляр класса `Maze`, который будет собираться другими операциями и в итоге будет возвращен клиенту (с помощью `GetMaze`):

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

Операция `BuildRoom` создает комнату и строит вокруг нее стены:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Чтобы построить дверь между двумя комнатами, `StandardMazeBuilder` находит обе комнаты в лабиринте и их общую стену:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

Теперь для создания лабиринта клиенты могут использовать `CreateMaze` в сочетании с `StandardMazeBuilder`:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

Мы могли бы поместить все операции класса `StandardMazeBuilder` в класс `Maze` и позволить каждому лабиринту строить самого себя. Но чем меньше класс `Maze`, тем проще в нем разобраться и внести изменения, а `StandardMazeBuilder` легко отделяется от `Maze`. Еще важнее то, что разделение этих двух классов позволяет создать множество разновидностей класса `MazeBuilder`, в каждом из которых есть собственные классы для комнат, дверей и стен.

Необычной разновидностью `MazeBuilder` является класс `CountingMazeBuilder`. Он вообще не создает никакого лабиринта, а лишь подсчитывает число компонентов разного вида, которые могли бы быть созданы:

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

Конструктор инициализирует счетчики, а замещенные операции класса `MazeBuilder` увеличивают их:

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

А вот как клиент мог бы использовать класс `CountingMazeBuilder`:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "В лабиринте есть "
     << rooms << " комнат и "
     << doors << " дверей" << endl;
```

Известные применения

Приложение для конвертирования из формата RTF взято из библиотеки ET++ [WGM88]. Строитель используется в ней для обработки текста, хранящегося в формате RTF.

Паттерн строитель широко применяется в языке Smalltalk80 [Par90]:

- класс `Parser` в подсистеме компиляции — это распорядитель, которому в качестве аргумента передается объект `ProgramNodeBuilder`. Объект класса `Parser` извещает объект `ProgramNodeBuilder` после распознавания каждой синтаксической конструкции. После завершения разбора `Parser` обращается к строителю за созданным деревом разбора и возвращает его клиенту;
- `ClassBuilder` — это строитель, которым пользуются все классы для создания своих подклассов. В данном случае этот класс выступает одновременно в качестве распорядителя и продукта;
- `ByteCodeStream` — это строитель, который создает откомпилированный метод в виде массива байт. `ByteCodeStream` является примером нестандартного применения паттерна строитель, поскольку сложный объект представляется в виде массива байт, а не обычного объекта Smalltalk. Но интерфейс к `ByteCodeStream` типичен для строителя, и этот класс легко можно было бы заменить другим, который представляет программу в виде составного объекта.

Каркас Service Configurator из Adaptive Communications Environment использует строителя для построения компонентов сетевых служб, связанных с сервером во время выполнения [SS94]. Описание компонентов формулируется на языке конфигурации, который разбирается парсером LALR(1). Семантические действия парсера выполняют со строителем операции, добавляющие информацию в компонент службы. В данном случае парсер выполняет функции распорядителя.

Родственные паттерны

Абстрактная фабрика (113) похожа на строителя в том смысле, что может конструировать сложные объекты. Основное различие между ними в том, что строитель специализируется на пошаговом конструировании объекта, а абстрактная фабрика — на создании семейств объектов (простых или сложных). Строитель возвращает продукт на последнем шаге, тогда как с точки зрения абстрактной фабрики продукт возвращается немедленно.

Паттерн компоновщик (196) — это то, что часто создает строитель.

ПАТТЕРН FACTORY METHOD (ФАБРИЧНЫЙ МЕТОД)

Название и классификация паттерна

Фабричный метод — паттерн, порождающий классы.

Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, экземпляры какого класса должны создаваться. Фабричный метод позволяет классу делегировать создание экземпляров подклассам.

Другие названия

`Virtual Constructor` (виртуальный конструктор).

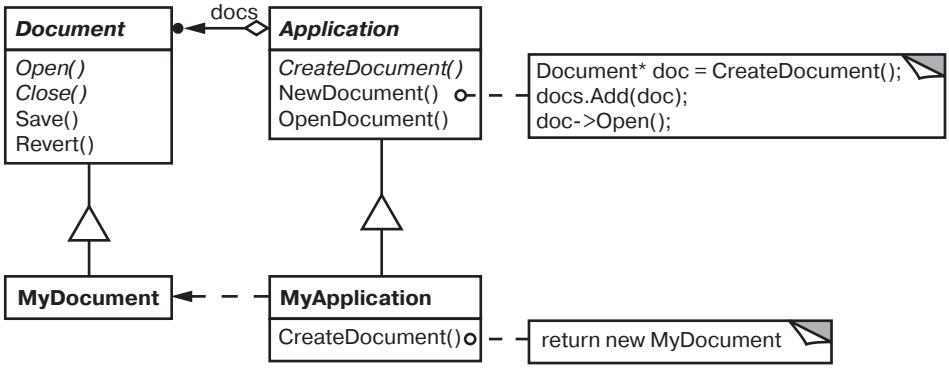
Мотивация

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе — это классы `Application` и `Document`. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определим классы `DrawingApplication` и `DrawingDocument`. Класс `Application` отвечает за управление документами и создает их по мере необходимости — допустим, когда пользователь выбирает из меню пункт `Open` (открыть) или `New` (создать).

Поскольку выбор подкласса `Document` для создания экземпляра зависит от приложения, то `Application` не может «предсказать», что именно понадобится. Этот класс знает только то, *когда* нужно создать экземпляр нового документа, а не *какой* документ создать. Возникает дилемма: каркас должен создавать экземпляры класса, но «знает» он лишь об абстрактных классах, экземпляры которых создать нельзя.

Проблема решается при помощи паттерна фабричный метод. Он инкапсулирует информацию о том, какой подкласс класса `Document` должен создаваться, и выводит это знание за пределы каркаса.



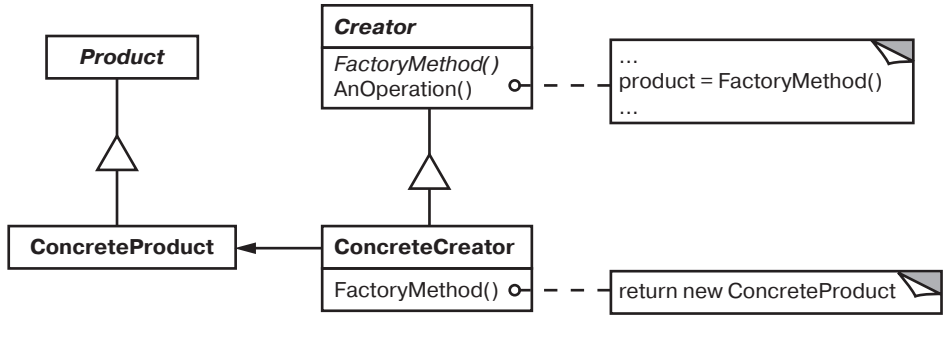
Подклассы класса `Application` переопределяют абстрактную операцию `CreateDocument` таким образом, чтобы она возвращала подходящий подкласс класса `Document`. Как только будет создан экземпляр подкласса `Application`, он может создавать экземпляры специфических для приложения документов, ничего не зная об их классах. Операция `CreateDocument` называется *фабричным методом*, так как она отвечает за «изготовление» объекта.

Применимость

Основные условия для применения паттерна фабричный метод:

- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, определялись подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вам нужно локализовать информацию о том, какой класс принимает эти обязанности на себя.

Структура



Участники

- **Product** (`Document`) — продукт:
  - определяет интерфейс объектов, создаваемых фабричным методом;
- **ConcreteProduct** (`MyDocument`) — конкретный продукт:
  - реализует интерфейс `Product`;
- **Creator** (`Application`) — создатель:
  - объявляет фабричный метод, возвращающий объект типа Product. `Creator` может также определять реализацию по умолчанию фабричного метода, который возвращает объект `ConcreteProduct`;
  - может вызывать фабричный метод для создания объекта `Product`.
- **ConcreteCreator** (`MyApplication`) — конкретный создатель:
  - замещает фабричный метод, возвращающий объект `ConcreteProduct`.

Отношения

Создатель полагается на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Результаты

Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса `Product`, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

Потенциальный недостаток фабричного метода состоит в том, что клиентам, возможно, придется создавать подкласс класса `Creator` для создания лишь одного объекта `ConcreteProduct`. Порождение подклассов оправданно, если клиенту так или иначе приходится создавать подклассы `Creator`, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.

А вот еще два последствия применения паттерна фабричный метод:

- *подклассам предоставляются операции-зацепки (hooks).* Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки для предоставления расширенной версии объекта.

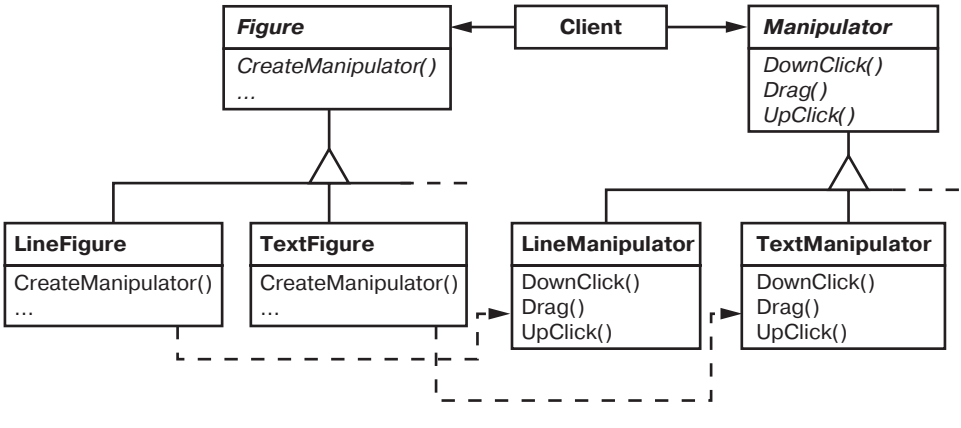
  В примере с документом класс `Document` мог бы определить фабричный метод `CreateFileDialog`, который создает диалоговое окно по умолчанию

для открытия существующего документа. Подкласс этого класса мог бы определить специализированное для приложения диалоговое окно, заместив этот фабричный метод. В данном случае фабричный метод не является абстрактным, а содержит разумную реализацию по умолчанию;

- *соединение параллельных иерархий*. В примерах, которые мы рассматривали до сих пор, фабричные методы вызывались только создателем. Но это совершенно необязательно: клиенты тоже могут применять фабричные методы, особенно при наличии параллельных иерархий классов.

  Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, который не является производным от него. Рассмотрим, например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши. Реализовать такие взаимодействия бывает непросто; часто приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но это состояние используется только во время самой манипуляции, поэтому помещать его в объект, представляющий фигуру, не следует. К тому же фигуры по-разному ведут себя, когда пользователь манипулирует ими. Например, растягивание отрезка может сводиться к изменению положения концевой точки, а растягивание текста — к изменению междустрочных интервалов.

  При таких ограничениях лучше использовать отдельный объект-манипулятор `Manipulator`, который реализует взаимодействие и контролирует его текущее состояние. Разным фигурам будут соответствовать разные манипуляторы, являющиеся подклассом `Manipulator`. Получающаяся иерархия класса `Manipulator` параллельна (по крайней мере частично) иерархии класса `Figure`.



Класс `Figure` предоставляет фабричный метод `CreateManipulator`, который позволяет клиентам создавать соответствующий фигуре манипулятор. Подклассы `Figure` замещают этот метод так, чтобы он возвращал подходящий для них подкласс `Manipulator`. Вместо этого класс `Figure` может реализовать `CreateManipulator` так, что он будет возвращать экземпляр класса `Manipulator` по умолчанию, а подклассы `Figure` могут наследовать это умолчание. Те классы фигур, которые работают по описанному принципу, не нуждаются в специальном манипуляторе, поэтому иерархии параллельны только отчасти.

Обратите внимание, как фабричный метод определяет связь между обеими иерархиями классов. В нем локализуется знание о том, какие классы способны работать совместно.

Реализация

Рассмотрим следующие вопросы, возникающие при использовании паттерна фабричный метод:

- *две основных разновидности паттерна*: (1) случай, когда класс `Creator` является абстрактным и не содержит реализации объявленного в нем фабричного метода, и (2) `Creator` — конкретный класс, в котором по умолчанию есть реализация фабричного метода. Редко, но встречается и абстрактный класс, имеющий реализацию по умолчанию.

  В первом случае для определения реализации *необходимы* подклассы, поскольку никакой разумной реализации по умолчанию не существует. При этом обходится проблема, связанная с необходимостью создания экземпляров заранее неизвестных классов. Во втором случае конкретный класс `Creator` использует фабричный метод, главным образом ради повышения гибкости. Происходящее соответствует правилу: «Создавай объекты в отдельной операции, чтобы подклассы могли подменить способ их создания». Соблюдение этого правила гарантирует, что авторы подклассов смогут при необходимости изменить класс объектов, экземпляры которых создаются их родителем;

- *параметризованные фабричные методы*. Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который определяет вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий интерфейс `Product`. В примере с документами класс `Application` может поддерживать разные виды документов. Вы передаете методу `CreateDocument` лишний параметр, который и определяет, документ какого вида нужно создать.

В каркасе Unidraw для создания графических редакторов [VL90] используется именно этот подход для реконструкции объектов, сохраненных на диске. Unidraw определяет класс `Creator` с фабричным методом `Create`, которому в аргументе передается идентификатор класса, определяющий, экземпляр какого класса должен создаваться. Когда Unidraw сохраняет объект на диске, он сначала записывает идентификатор класса, а затем его переменные экземпляра. При реконструкции объекта сначала читается идентификатор класса.

После чтения идентификатора класса каркас вызывает операцию `Create`, передавая ей этот идентификатор как параметр. `Create` ищет конструктор соответствующего класса и с его помощью создает экземпляр. И наконец, `Create` вызывает операцию `Read` созданного объекта, которая считывает с диска остальную информацию и инициализирует переменные экземпляра.

Параметризованный фабричный метод в общем случае имеет следующий вид (здесь `MyProduct` и `YourProduct` — подклассы `Product`):

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE)  return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // Повторить для всех остальных продуктов...

    return 0;
}
```

Замещение параметризованного фабричного метода позволяет легко и избирательно расширить или заменить продукты, которые изготавливает создатель. Можно завести новые идентификаторы для новых видов продуктов или ассоциировать существующие идентификаторы с другими продуктами.

Например, подкласс `MyCreator` мог бы переставить местами `MyProduct` и `YourProduct` для поддержки третьего подкласса `TheirProduct`:

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS)  return new MyProduct;
    if (id == MINE)   return new YourProduct;
        // N.B.: YOURS и MINE переставлены
```

```
    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id); // Вызывается, если других
                                // вариантов не осталось
}
```

Обратите внимание, что в самом конце операция вызывает метод `Create` родительского класса. Это происходит из-за того, что `MyCreator::Create` обрабатывает только продукты `YOURS`, `MINE` и `THEIRS` иначе, чем родительский класс. Другие классы его не интересуют. По этой причине `MyCreator` *расширяет* некоторые виды создаваемых продуктов, а создание остальных поручает своему родительскому классу;

- *вариации и проблемы, зависящие от конкретного языка*. В разных языках возникают собственные интересные варианты и некоторые нюансы.

  Так, в программах на Smalltalk часто используется метод, который возвращает класс создаваемого экземпляра. Фабричный метод `Creator` может воспользоваться возвращенным значением для создания продукта, а `ConcreteCreator` может сохранить или даже вычислить это значение. В результате привязка к типу конкретного создаваемого продукта `ConcreteProduct` происходит еще позже.

  В версии примера `Document` на языке Smalltalk метод `documentClass` может определяться в классе `Application`. Этот метод возвращает подходящий класс `Document` для создания экземпляров документов. Реализация метода `documentClass` в классе `MyApplication` возвращает класс `MyDocument`. Таким образом, в классе `Application` мы имеем

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

  А в классе `MyApplication`:

```
documentClass
    ^ MyDocument
```

  с возвращением класса `MyDocument`, экземпляр которого должен быть создан, приложению Application.

  Еще более гибкий подход, сходный с параметризованными фабричными методами, основан на сохранении создаваемого класса в переменной

класса `Application`. В таком случае для изменения продукта не нужно будет порождать подкласс `Application`.

В C++ фабричные методы всегда являются виртуальными функциями, а часто даже чисто виртуальными. Нужно быть осторожней и не вызывать фабричные методы в конструкторе класса `Creator`: в этот момент фабричный метод в производном классе `ConcreteCreator` еще недоступен.

Проблему можно обойти, если обращаться к продуктам только с помощью функций доступа, создающих продукт по запросу. Вместо того чтобы создавать конкретный продукт, конструктор просто инициализирует его нулем. Функция доступа возвращает продукт, но сначала проверяет, что он существует (и если не существует — создает продукт). Подобный подход часто называют *отложенной инициализацией*. В следующем примере показана типичная реализация:

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}
```

- *использование шаблонов, чтобы не порождать подклассы*. К сожалению, возможна ситуация, когда вам придется порождать подклассы только для того, чтобы создать подходящие объекты-продукты. В C++ этого можно избежать, предоставив шаблонный подкласс класса `Creator`, параметризованный классом `Product`:

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
```

```
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

С таким шаблоном клиент передает только класс продукта, порождать подклассы от `Creator` не требуется:

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

- *соглашения об именах*. Рекомендуется применять такие соглашения об именах, которые дают ясно понять, что вы пользуетесь фабричными методами. Например, каркас MacApp на платформе Macintosh [App89] всегда объявляет абстрактную операцию, которая определяет фабричный метод, в виде `Class* DoMakeClass()`, где `Class` — это класс продукта.

Пример кода

Функция `CreateMaze` строит и возвращает лабиринт. Одна из проблем, связанных с этой функцией, состоит в том, что классы лабиринта, комнат, дверей и стен жестко «зашиты» в данной функции. Мы введем фабричные методы, которые позволят выбирать эти компоненты подклассам.

Сначала определим фабричные методы в игре `MazeGame` для создания объектов лабиринта, комнат, дверей и стен:

```
class MazeGame {
public:
    Maze* CreateMaze();

// фабричные методы:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
```

```
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Каждый фабричный метод возвращает один из компонентов лабиринта. Класс `MazeGame` предоставляет реализации по умолчанию, которые возвращают простейшие варианты лабиринта, комнаты, двери и стены.

Теперь мы можем переписать функцию `CreateMaze` с использованием этих фабричных методов:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

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

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

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

    return aMaze;
}
```

В играх могут порождаться различные подклассы `MazeGame` для специализации частей лабиринта. Эти подклассы могут переопределять некоторые (или все) методы, от которых зависят разновидности продуктов. Например, в игре `BombedMazeGame` продукты `Room` и `Wall` могут быть переопределены так, чтобы возвращать комнату и стену с заложенной бомбой:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
```

```
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

Вариация `EnchantedMazeGame` может определяться так:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Известные применения

Фабричные методы в изобилии встречаются в инструментальных библиотеках и каркасах. Рассмотренный выше пример с документами — это типичное применение в каркасе MacApp и библиотеке ET++ [WGM88]. Пример с манипулятором заимствован из каркаса Unidraw.

Класс `View` в схеме «модель — представление — контроллер» из языка Smalltalk80 имеет метод `defaultController`, который создает контроллер, и этот метод выглядит как фабричный [Par90]. Но подклассы `View` задают класс своего контроллера по умолчанию, определяя метод `defaultControllerClass`, возвращающий класс, экземпляры которого создает `defaultController`. Таким образом, реальным фабричным методом является `defaultControllerClass`, то есть метод, который должен переопределяться в подклассах.

Более необычным является пример фабричного метода `parserClass`, тоже взятый из Smalltalk80, который определяется поведением `Behavior` (суперкласс всех объектов, представляющих классы). Он позволяет классу использовать специализированный анализатор своего исходного кода. Например, клиент может определить класс `SQLParser` для анализа исходного кода класса, содержащего встроенные команды на языке SQL. Класс `Behavior` реализует `parserClass` так, что тот возвращает стандартный для Smalltalk класс анализатора `Parser`. Класс же, включающий предложения SQL, замещает этот метод (как метод класса) и возвращает класс `SQLParser`.

Система Orbix ORB от компании IONA Technologies [ION94] использует фабричный метод для генерирования подходящих заместителей (см. паттерн заместитель) в случае, когда объект запрашивает ссылку на удаленный объект. Фабричный метод позволяет без труда заменить подразумеваемого заместителя, например таким, который применяет кэширование на стороне клиента.

Родственные паттерны

Абстрактная фабрика (113) часто реализуется с помощью фабричных методов. Пример в разделе «Мотивация» из описания абстрактной фабрики иллюстрирует также и паттерн фабричный метод.

Паттерн фабричный метод часто вызывается внутри шаблонных методов (373). В примере с документами `NewDocument` — это шаблонный метод.

Прототипы (146) не нуждаются в порождении подклассов от `Creator`. Однако им часто бывает необходима операция `Initialize` в классе `Product`. `Creator` использует `Initialize` для инициализации объекта. Фабричному методу такая операция не требуется.

ПАТТЕРН PROTOTYPE (ПРОТОТИП)

Название и классификация паттерна

Прототип — паттерн, порождающий объекты.

Назначение

Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Мотивация

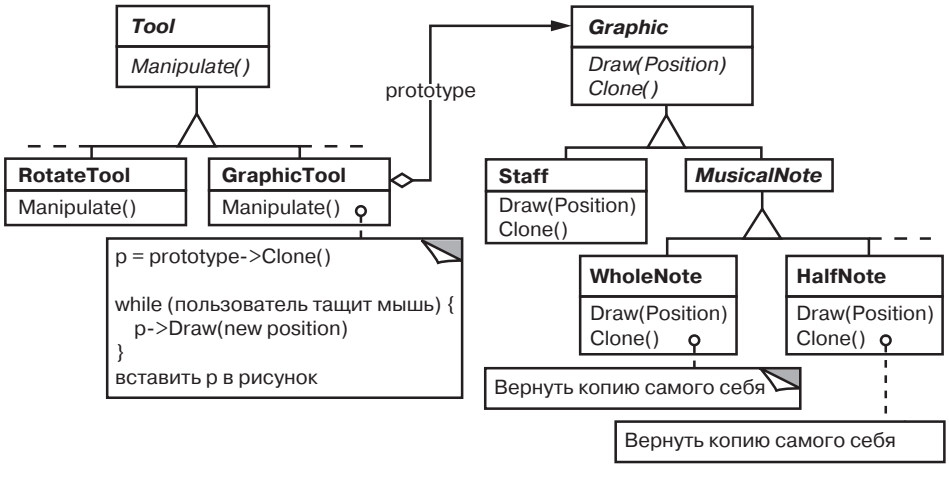
Построить музыкальный редактор удалось бы путем адаптации общего каркаса графических редакторов и добавления новых объектов, представляющих ноты, паузы и нотный стан. В каркасе редактора может присутствовать палитра инструментов для добавления в партитуру этих музыкальных объектов. Палитра может также содержать инструменты для выбора, перемещения и иных манипуляций с объектами. Так, щелкая, например, по значку четверти, пользователь поместил бы тем самым четвертные ноты в партитуру. Или, применив инструмент перемещения, сдвигал бы ноту на стане вверх или вниз, чтобы изменить ее высоту.

Предположим, что каркас предоставляет абстрактный класс `Graphic` для графических компонентов вроде нот и нотных станов, а также абстрактный класс `Tool` для определения инструментов в палитре. Кроме того, в каркасе имеется предопределенный подкласс `GraphicTool` для инструментов, которые создают графические объекты и добавляют их в документ.

Однако класс `GraphicTool` создает проблему для проектировщика каркаса. Классы нот и нотных станов специфичны для нашего приложения, а класс `GraphicTool` принадлежит каркасу. Этому классу ничего неизвестно о том, как создавать экземпляры наших музыкальных классов и добавлять их в партитуру. Можно было бы породить от `GraphicTool` подклассы для каждого вида музыкальных объектов, но тогда оказалось бы слишком много классов, отличающихся только тем, какой музыкальный объект они создают. Мы знаем, что гибкой альтернативой порождению подклассов является композиция. Вопрос в том, как каркас мог бы воспользоваться ею для параметризации экземпляров `GraphicTool` *классом* того объекта `Graphic`, который предполагается создать.

Решение — заставить `GraphicTool` создавать новый графический объект копированием или «клонированием» экземпляра подкласса класса `Graphic`. Этот экземпляр мы будем называть *прототипом*. `GraphicTool` параметризуется прототипом, который он должен клонировать и добавить в документ. Если все подклассы `Graphic` поддерживают операцию `Clone`, то `GraphicTool` может клонировать любой вид графических объектов.

Итак, в нашем музыкальном редакторе каждый инструмент для создания музыкального объекта — это экземпляр класса `GraphicTool`, инициализированный тем или иным прототипом. Любой экземпляр `GraphicTool` будет создавать музыкальный объект, клонируя его прототип и добавляя клон в партитуру.



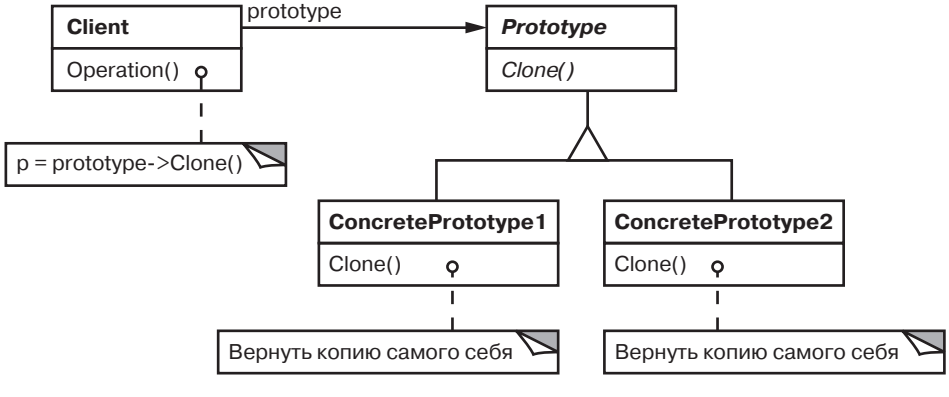
Использование паттерна прототип позволит еще больше сократить число классов. Для целых и половинных нот у нас есть отдельные классы, но, быть может, это излишне. Вместо этого они могли бы быть экземплярами одного и того же класса, инициализированного разными растровыми изображениями и длительностями звучания. Инструмент для создания целых нот становится просто объектом класса `GraphicTool`, в котором прототип `MusicalNote` инициализирован целой нотой. Это может значительно уменьшить число классов в системе. Заодно упрощается добавление нового вида нот в музыкальный редактор.

Применимость

Используйте паттерн прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты; *кроме того*:

- классы для создания экземпляров определяются во время выполнения, например с помощью динамической загрузки; *или*
- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов; *или*
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может быть удобнее установить соответствующее число прототипов и клонировать их, а не создавать экземпляр каждый раз вручную в подходящем состоянии.

Структура



Участники

- **Prototype** (`Graphic`) — прототип:
  - объявляет интерфейс для клонирования самого себя;

- **ConcretePrototype** (`Staff` — нотный стан, `WholeNote` — целая нота, `HalfNote` — половинная нота) — конкретный прототип:
  - реализует операцию клонирования себя;
- **Client** (`GraphicTool`) — клиент:
  - создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Отношения

Клиент обращается к прототипу, чтобы тот создал свою копию.

Результаты

У прототипа те же самые результаты, что у абстрактной фабрики (113) и строителя (124): он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать с классами, специфичными для приложения, без модификаций.

Ниже перечислены дополнительные преимущества паттерна прототип:

- *добавление и удаление продуктов во время выполнения*. Прототип позволяет включать новый конкретный класс продуктов в систему, просто зарегистрировав новый экземпляр-прототип на стороне клиента. Это несколько более гибкое решение по сравнению с тем, что удастся сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения;
- *определение новых объектов путем изменения значений*. Динамичные системы позволяют определять поведение посредством композиции объектов — например, путем задания значений переменных объекта, — а не посредством определения новых классов. Фактически вы определяете новые виды объектов, создавая экземпляры уже существующих классов и регистрируя их экземпляры как прототипы клиентских объектов. Клиент может изменить поведение, делегируя свои обязанности прототипу.

  Такой дизайн позволяет пользователям определять новые «классы» без программирования. Фактически клонирование объекта аналогично созданию экземпляра. Паттерн прототип может резко уменьшить число необходимых системе классов. В нашем музыкальном редакторе с помощью одного только класса `GraphicTool` удастся создать бесконечное разнообразие музыкальных объектов;

- *определение новых объектов путем изменения структуры*. Многие приложения строят объекты из крупных и мелких составляющих. Например, редакторы для проектирования печатных плат создают электрические схемы из подсхем[1]. Такие приложения часто позволяют создавать экземпляры сложных, определенных пользователем структур — скажем, для многократного использования некоторой подсхемы.

  Паттерн прототип поддерживает и такую возможность. Мы просто добавляем подсхему как прототип в палитру доступных элементов схемы. При условии что объект, представляющий составную схему, реализует операцию `Clone` как глубокое копирование, схемы с разными структурами могут выступать в качестве прототипов;

- *уменьшение числа подклассов*. Паттерн фабричный метод часто порождает иерархию классов `Creator`, параллельную иерархии классов продуктов. Прототип позволяет клонировать прототип, а не запрашивать фабричный метод создать новый объект. Поэтому иерархия класса `Creator` становится вообще ненужной. Это преимущество касается главным образом языков типа C++, где классы не рассматриваются как настоящие объекты. В языках же типа Smalltalk и Objective C это не так существенно, поскольку всегда можно использовать объект-класс в качестве создателя. В таких языках объекты-классы уже выступают как прототипы;

- *динамическая настройка конфигурации приложения классами*. Некоторые среды позволяют динамически загружать классы в приложение во время его выполнения. Паттерн прототип — это ключ к применению таких возможностей в языке типа C++.

  Приложение, которое создает экземпляры динамически загружаемого класса, не может обращаться к его конструктору статически. Вместо этого исполняющая среда автоматически создает экземпляр каждого класса в момент его загрузки и регистрирует экземпляр в диспетчере прототипов (см. раздел «Реализация»). Затем приложение может запросить у диспетчера прототипов экземпляры вновь загруженных классов, которые изначально не были связаны с программой. Каркас приложений ET++ [WGM88] в своей исполняющей среде использует именно такую схему.

Основной недостаток паттерна прототип заключается в том, что каждый подкласс класса `Prototype` должен реализовывать операцию `Clone`, а это далеко не всегда просто. Например, сложно добавить операцию `Clone`, если рассматриваемые классы уже существуют. Проблемы возникают

[1] Для таких приложений характерны паттерны компоновщик и декоратор.

и в случае, если во внутреннем представлении объекта присутствуют другие объекты, не поддерживающие копирования, или наличествуют циклические ссылки.

Реализация

Прототип особенно полезен в статически типизированных языках вроде C++, где классы не являются объектами, а во время выполнения информации о типе недостаточно или нет вовсе. Меньший интерес данный паттерн представляет для таких языков, как Smalltalk или Objective C, в которых и так уже есть нечто эквивалентное прототипу (а именно — объект-класс) для создания экземпляров каждого класса. Этот паттерн уже встроен в языки, основанные на прототипах, например Self [US87], где создание любого объекта выполняется путем клонирования прототипа.

Рассмотрим основные вопросы, возникающие при реализации прототипов:

- *использование диспетчера прототипов*. Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), создайте реестр доступных прототипов. Клиенты должны не управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Такой реестр мы будем называть *диспетчером прототипов*.

  Диспетчер прототипов — это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и просматривать реестр во время выполнения — а значит, расширять систему и вести контроль над ее состоянием без написания кода;

- *реализация операции Clone*. Самая трудная часть паттерна прототип — правильная реализация операции `Clone`. Особенно сложно это в случае, когда в структуре объекта есть циклические ссылки.

  В большинстве языков имеется некоторая поддержка для клонирования объектов. Например, Smalltalk предоставляет реализацию копирования, которую все подклассы наследуют от класса `Object`. В C++ поддерживаются копирующие конструкторы. Тем не менее, эти средства не решают проблему «глубокого и поверхностного копирования» [GR83]. Суть ее в следующем: должны ли при клонировании объекта клонироваться также и его переменные экземпляра или клон просто совместно использует с оригиналом эти переменные?

Поверхностное копирование просто реализуется, и часто его бывает достаточно. В частности, его предоставляет по умолчанию Smalltalk. В C++ копирующий конструктор по умолчанию выполняет копирование на уровне членов класса, то есть указатели совместно используются копией и оригиналом. Но для клонирования прототипов со сложной структурой обычно необходимо глубокое копирование, поскольку клон должен быть независим от оригинала. Поэтому нужно гарантировать, что компоненты клона являются клонами компонентов прототипа. При клонировании вам приходится решать, какие компоненты могут использоваться совместно (и могут ли вообще).

Если объекты в системе предоставляют операции `Save` (сохранить) и `Load` (загрузить), то разрешается воспользоваться ими для реализации операции `Clone` по умолчанию, просто сохранив и сразу же загрузив объект. Операция `Save` сохраняет объект в буфере, находящемся в памяти, а `Load` создает дубликат, реконструируя объект из буфера;

- *инициализация клонов*. Хотя некоторым клиентам вполне достаточно клона как такового, другим нужно полностью или частично инициализировать его внутреннее состояние. Обычно передать начальные значения операции `Clone` невозможно, поскольку их число различно для разных классов прототипов. Одним прототипам нужно много параметров инициализации, другим вообще ничего не требуется. Передача `Clone` параметров мешает построению единообразного интерфейса клонирования.

  Возможно, в ваших классах прототипов уже определяются операции для установки и сброса некоторых важных элементов состояния. Если так, то этими операциями можно воспользоваться сразу после клонирования. В противном случае, возможно, понадобится ввести операцию `Initialize` (см. раздел «Пример кода»), которая принимает начальные значения в качестве аргументов и соответственно устанавливает внутреннее состояние клона. Будьте осторожны, если операция `Clone` реализует глубокое копирование: возможно, копии придется удалять (явно или внутри `Initialize`) перед повторной инициализацией.

Пример кода

Мы определим подкласс `MazePrototypeFactory` класса `MazeFactory`. Этот подкласс будет инициализироваться прототипами объектов, которые ему предстоит создавать, поэтому нам не придется порождать подклассы только ради изменения классов создаваемых стен или комнат.

`MazePrototypeFactory` дополняет интерфейс `MazeFactory` конструктором, в аргументах которого передаются прототипы:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Новый конструктор просто инициализирует свои прототипы:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
   _prototypeMaze = m;
   _prototypeWall = w;
   _prototypeRoom = r;
   _prototypeDoor = d;
}
```

Функции для создания стен, комнат и дверей похожи друг на друга: каждая клонирует, а затем инициализирует прототип. Определения функций `MakeWall` и `MakeDoor` выглядят так:

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

`MazePrototypeFactory` можно применить для создания лабиринта-прототипа или лабиринта по умолчанию, просто инициализируя его прототипами базовых компонентов:

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

Для изменения типа лабиринта следует инициализировать `MazePrototypeFactory` другим набором прототипов. Следующий вызов создает лабиринт с дверью типа `BombedDoor` и комнатой типа `RoomWithABomb`:

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

Объект, который предполагается использовать в качестве прототипа, например экземпляр класса `Wall`, должен поддерживать операцию `Clone`. Кроме того, у него должен быть копирующий конструктор для клонирования. Также может потребоваться операция для повторной инициализации внутреннего состояния. Мы добавим в класс `Door` операцию `Initialize`, чтобы дать клиентам возможность инициализировать комнаты клона.

Сравните следующее определение `Door` с приведенным на с. 111:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
```

```
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

Подкласс `BombedWall` должен заместить операцию `Clone` и реализовать соответствующий копирующий конструктор:

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Операция `BombedWall::Clone` возвращает `Wall*`, а ее реализация — указатель на новый экземпляр подкласса, то есть `BombedWall*`. Мы определяем `Clone` в базовом классе именно таким образом, чтобы клиентам, клонирующим прототип, не надо было знать о его конкретных подклассах. Клиентам никогда не придется приводить значение, возвращаемое `Clone`, к нужному типу.

В Smalltalk стандартный метод копирования, унаследованный от класса `Object`, может использоваться для клонирования любого прототипа `MapSite`. Фабрикой `MazeFactory` можно воспользоваться для изготовления любых необходимых прототипов — например, создать комнату по ее номеру `#room`. В классе `MazeFactory` есть словарь, связывающий имена с прототипами. Его метод `make:` выглядит так:

```
make: partName
    ^ (partCatalog at: partName) copy
```

Имея подходящие методы для инициализации `MazeFactory` прототипами, можно было бы создать простой лабиринт с помощью следующего кода:

```
CreateMaze
    on: (MazeFactory new
        with: Door new named: #door;
        with: Wall new named: #wall;
        with: Room new named: #room;
        yourself)
```

где определение метода класса `on:` для `CreateMaze` имеет вид

```
on: aFactory
    | room1 room2 |
    room1 := (aFactory make: #room) location: 1@1.
    room2 := (aFactory make: #room) location: 2@1.
    door := (aFactory make: #door) from: room1 to: room2.

    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: door;
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: (aFactory make: #wall).
    room2
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: (aFactory make: #wall);
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: door.
    ^ Maze new
        addRoom: room1;
        addRoom: room2;
        yourself
```

Известные применения

Вероятно, паттерн прототип был впервые использован в системе Sketchpad Айвена Сазерленда (Ivan Sutherland) [Sut63]. Первым широко известным применением этого паттерна в объектно-ориентированном языке была система ThingLab, в которой пользователи могли сформировать составной объект, а затем превратить его в прототип, поместив в библиотеку повторно используемых объектов [Bor81]. Адель Голдберг и Давид Робсон упоминают прототипы в качестве паттернов в работе [GR83], но Джеймс Коплиен [Cop92] рассматривает этот вопрос гораздо шире. Он описывает связанные с прототипом идиомы языка C++ и приводит много примеров и вариантов.

Etgdb — это оболочка для отладчиков на базе ET++, поддерживающая интерфейс типа point-and-click (укажи и щелкни) для различных командных отладчиков. Для каждого из них есть свой подкласс `DebuggerAdaptor`. Например, `GdbAdaptor` настраивает etgdb на синтаксис команд GNU gdb, а `SunDbxAdaptor` — на отладчик dbx компании Sun. Набор подклассов `DebuggerAdaptor` не «зашит» в etgdb. Вместо этого он получает имя адаптера из переменной среды, ищет в глобальной таблице прототип с указанным именем, а затем его клонирует. Чтобы добавить к etgdb новые отладчики, следует связать их с подклассом `DebuggerAdaptor`, разработанным для этого отладчика.

Библиотека приемов взаимодействия в программе Mode Composer хранит прототипы объектов, поддерживающих различные способы интерактивных отношений [Sha90]. Любой созданный с помощью Mode Composer способ взаимодействия можно применить в качестве прототипа, если поместить его в библиотеку. Паттерн прототип позволяет программе поддерживать неограниченное число вариантов отношений.

Пример музыкального редактора, обсуждавшийся в начале этого раздела, основан на каркасе графических редакторов Unidraw [VL90].

Родственные паттерны

В некоторых отношениях прототип и абстрактная фабрика (113) являются конкурентами, о чем будет рассказано в конце главы. Тем не менее, они могут использоваться совместно. Абстрактная фабрика может хранить набор прототипов, которые клонируются и возвращают изготовленные объекты.

В тех проектах, где активно применяются паттерны компоновщик (196) и декоратор (209), тоже можно извлечь пользу из прототипа.

ПАТТЕРН SINGLETON (ОДИНОЧКА)

Название и классификация паттерна

Одиночка — паттерн, порождающий объекты.

Назначение

Гарантирует, что у класса существует только один экземпляр, и предоставляет к нему глобальную точку доступа.

Мотивация

Для некоторых классов важно, чтобы существовал только один экземпляр. В системе может быть много принтеров, но может существовать лишь один спулер. В операционной системе должна быть только одна файловая система и единственный оконный менеджер. В цифровом фильтре может находиться только один аналого-цифровой преобразователь (АЦП). Бухгалтерская система обслуживает только одну компанию.

Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает создать несколько экземпляров класса.

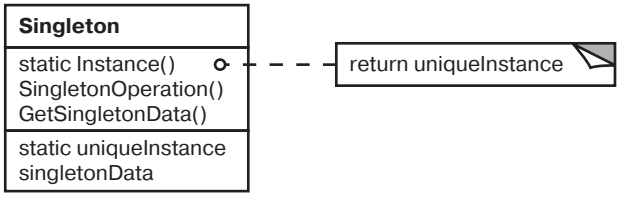
Более удачное решение — возложить на сам класс ответственность за то, что у него существует только один экземпляр. Класс может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Это и есть назначение паттерна одиночка.

Применимость

Основные условия для применения паттерна одиночка:

- должен существовать ровно один экземпляр некоторого класса, к которому может обратиться любой клиент через известную точку доступа;
- единственный экземпляр должен расширяться путем порождения подклассов, а клиенты должны иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Участники

- `Singleton` — одиночка:
  - определяет операцию `Instance`, которая позволяет клиентам получить доступ к единственному экземпляру. `Instance` — это операция класса,

то есть метод класса в терминологии Smalltalk и статическая функция класса в C++;

- может нести ответственность за создание собственного уникального экземпляра.

Отношения

Клиенты получают доступ к экземпляру класса `Singleton` только через его операцию `Instance`.

Результаты

Паттерн одиночка обладает рядом достоинств:

- *контролируемый доступ к единственному экземпляру*. Поскольку класс `Singleton` инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- *сокращение пространства имен*. Паттерн одиночка — шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;
- *возможность уточнения операций и представления*. От класса `Singleton` можно порождать подклассы, а приложение легко настраивается экземпляром расширенного класса. Приложение можно настроить экземпляром нужного класса во время выполнения;
- *возможность использования переменного числа экземпляров*. Паттерн позволяет легко изменить решение и разрешить появление более одного экземпляра класса `Singleton`. Более того, тот же подход может использоваться для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса `Singleton`;
- *большая гибкость, чем у операций класса*. Другой способ реализации функциональности одиночки — использование операций класса, то есть статических функций класса в C++ и методов класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких экземпляров класса. Кроме того, статические функции классов в C++ не могут быть виртуальными, что делает невозможной их полиморфную замену в подклассах.

Реализация

При использовании паттерна одиночка надо рассмотреть следующие вопросы:

- *гарантии существования единственного экземпляра*. Паттерн одиночка устроен так, что тот единственный экземпляр, который имеется у класса, — самый обычный, но сам класс написан так, что больше одного экземпляра создать не удастся. Чаще всего для этого операция, создающая экземпляры, скрывается за операцией класса (то есть за статической функцией или методом класса), которая гарантирует создание не более одного экземпляра. Данная операция имеет доступ к переменной, где хранится уникальный экземпляр, и гарантирует инициализацию переменной этим экземпляром перед возвратом ее клиенту. При таком подходе можно не сомневаться, что одиночка будет создан и инициализирован перед первым использованием.

  В C++ операция класса определяется с помощью статической функции `Instance` класса `Singleton`. В этот класс также включена статическая переменная `_instance`, которая содержит указатель на уникальный экземпляр.

  Класс `Singleton` объявлен следующим образом:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

  Соответствующая реализация выглядит так:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

  Клиенты осуществляют доступ к одиночке исключительно через функцию `Instance`. Переменная `_instance` инициализируется нулем, а статическая функция `Instance` возвращает ее значение, инициализируя ее уникальным экземпляром, если в текущий момент оно равно 0. Функция `Instance` использует отложенную инициализацию: возвращаемое ей значение не создается и не сохраняется вплоть до момента первого обращения.

Обратите внимание, что конструктор защищенный. Клиент, который попытается создать экземпляр класса `Singleton` непосредственно, получит ошибку на этапе компиляции. Тем самым гарантируется, что будет создан только один экземпляр.

Далее, поскольку `_instance` — указатель на объект класса `Singleton`, то функция `Instance` может присвоить этой переменной указатель на любой подкласс данного класса. Применение возможности мы увидим в разделе «Пример кода».

О реализации в C++ стоит сказать особо. Недостаточно определить рассматриваемый паттерн как глобальный или статический объект, а затем полагаться на автоматическую инициализацию. Тому есть три причины:

- невозможно гарантировать, что в программе будет объявлен только один экземпляр статического объекта;
- у нас может не быть достаточно информации для создания экземпляра каждого одиночки во время статической инициализации. Одиночке могут потребоваться данные, вычисляемые позже, во время выполнения программы;
- в C++ не определяется порядок вызова конструкторов для глобальных объектов через границы единиц трансляции [ES90]. Это означает, что между одиночками не может существовать никаких зависимостей. Если они есть, то ошибок не избежать.

Еще один недостаток глобальных/статических объектов в том, что приходится создавать всех одиночек, даже если они не используются. Применение статической функции класса решает эту проблему.

В Smalltalk функция, возвращающая уникальный экземпляр, реализуется как метод класса Singleton. Чтобы гарантировать единственность экземпляра, следует заместить операцию `new`. Получающийся класс мог бы иметь два метода класса (в них `SoleInstance` — это переменная класса, которая больше нигде не используется):

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

- *Порождение подклассов Singleton.* Основной вопрос не столько в том, как определить подкласс, а в том, как оформить его уникальный экзем-

пляр, чтобы клиенты могли использовать его. По существу, переменная, ссылающаяся на экземпляр одиночки, должна инициализироваться вместе с экземпляром подкласса. Простейший способ добиться этого — определить одиночку, которого нужно применять в операции `Instance` класса `Singleton`. В разделе «Пример кода» показывается, как можно реализовать эту технику с помощью переменных среды.

Другой способ выбора подкласса `Singleton` — вынести реализацию операции `Instance` из родительского класса (например, `MazeFactory`) и поместить ее в подкласс. Это позволит программисту на C++ задать класс одиночки на этапе компоновки (например, скомпоновав программу с объектным файлом, содержащим другую реализацию), но от клиента одиночка будет по-прежнему скрыт.

Такой подход фиксирует выбор класса одиночки на этапе компоновки, затрудняя тем самым его подмену во время выполнения. Применение условных операторов для выбора подкласса увеличивает гибкость решения, но все равно множество возможных классов `Singleton` остается жестко «зашитым» в код. В общем случае ни тот, ни другой подход не обеспечивают достаточной гибкости.

Ее можно добиться за счет использования реестра одиночек. Вместо того чтобы задавать множество возможных классов `Singleton` в операции `Instance`, одиночки могут регистрировать себя по имени в некотором всем известном реестре.

Реестр сопоставляет одиночкам строковые имена. Когда операции `Instance` нужен некоторый одиночка, она запрашивает его у реестра по имени. Начинается поиск указанного одиночки, и, если он существует, реестр возвращает его. Такой подход освобождает `Instance` от необходимости «знать» все возможные классы или экземпляры `Singleton`. Нужен лишь единый для всех классов `Singleton` интерфейс, включающий операции с реестром:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Операция `Register` регистрирует экземпляр класса `Singleton` под указанным именем. Чтобы не усложнять реестр, мы будем хранить его в виде списка объектов `NameSingletonPair`. Каждый такой объект устанавливает соответствие между именем и одиночкой. Операция `Lookup` ищет одиночку по имени. Допустим, имя нужного одиночки передается в переменной среды:

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // Задается пользователем или средой при запуске

        _instance = Lookup(singletonName);
        // Lookup возвращает 0, если такой одиночка не найден.
    }
    return _instance;
}
```

В какой момент классы `Singleton` регистрируют себя? Одна из возможностей — конструктор. Например, подкласс `MySingleton` мог бы работать так:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Разумеется, конструктор не будет вызван, пока кто-то не создаст экземпляр класса, но ведь это та самая проблема, которую паттерн одиночка и пытается разрешить! В C++ ее можно попытаться обойти, определив статический экземпляр класса `MySingleton`. Например, можно вставить строку

```
static MySingleton theSingleton;
```

в файл с реализацией `MySingleton`.

Теперь класс `Singleton` не отвечает за создание одиночки. Его основной обязанностью становится обеспечение доступа к объекту-одиночке из любой части системы. Решение со статическим объектом по-прежнему имеет потенциальный недостаток: необходимость создания экземпляров всех возможных подклассов `Singleton`, без чего они не будут зарегистрированы.

Пример кода

Предположим, нам надо определить класс `MazeFactory` для создания лабиринтов, описанный на с. 111. `MazeFactory` определяет интерфейс для

построения различных частей лабиринта. В подклассах эти операции могут переопределяться, чтобы возвращать экземпляры специализированных классов продуктов, например объекты `BombedWall`, а не просто `Wall`.

Существенно здесь то, что приложению `Maze` нужен лишь один экземпляр фабрики лабиринтов и он должен быть доступен в коде, строящем любую часть лабиринта. Тут-то паттерн `одиночка` и приходит на помощь. Сделав фабрику `MazeFactory` одиночкой, мы сможем обеспечить глобальную доступность объекта, представляющего лабиринт, не прибегая к глобальным переменным.

Для простоты предположим, что мы никогда не порождаем подклассов от `MazeFactory`. (Чуть ниже будет рассмотрен альтернативный подход.) В C++ для того, чтобы превратить фабрику в одиночку, мы добавляем в класс `MazeFactory` статическую операцию `Instance` и статический член `_instance`, в котором будет храниться единственный экземпляр. Нужно также сделать конструктор защищенным, чтобы предотвратить случайное создание экземпляра, в результате которого будет создан лишний экземпляр:

```
class MazeFactory {
 public:
     static MazeFactory* Instance();

     // Здесь находится существующий интерфейс
 protected:
     MazeFactory();
 private:
     static MazeFactory* _instance;
 };
```

Соответствующая реализация выглядит так:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Теперь посмотрим, что случится, когда у `MazeFactory` есть подклассы и определяется, какой из них использовать. Вид лабиринта мы будем выбирать с помощью переменной среды, поэтому добавим код, который создает экземпляр нужного подкласса `MazeFactory` в зависимости от значения данной

переменной. Лучше всего поместить код в операцию `Instance`, поскольку она уже и так создает экземпляр `MazeFactory`:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ...другие возможные подклассы

        } else { // по умолчанию
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Отметим, что операцию `Instance` придется модифицировать при определении каждого нового подкласса `MazeFactory`. Возможно, в данном приложении это не создаст проблем, но для абстрактных фабрик, определенных в каркасе, такой подход трудно назвать приемлемым.

Одно из решений — воспользоваться принципом реестра, описанным в разделе «Реализация». Может помочь и динамическое связывание, тогда приложению не нужно будет загружать все неиспользуемые подклассы.

Известные применения

Примером паттерна одиночка в Smalltalk 80 [Par90] является множество изменений кода, представленное классом `ChangeSet`. Более тонкий пример — это отношение между классами и их *метаклассами*. Метаклассом называется класс класса, каждый метакласс существует в единственном экземпляре. У метакласса нет имени (разве что косвенное, определяемое экземпляром), но он контролирует свой уникальный экземпляр, и создать второй обычно не разрешается.

В библиотеке InterViews, предназначенной для создания пользовательских интерфейсов [LCI+92], паттерн одиночка применяется для доступа к единственным экземплярам классов `Session` (сессия) и `WidgetKit` (набор виджетов). Классом `Session` определяется главный цикл событий в приложении.

Он хранит пользовательские настройки стиля и управляет подключением к одному или нескольким физическим дисплеям. `WidgetKit` — это абстрактная фабрика (113) для определения внешнего облика интерфейсных виджетов. Операция `WidgetKit::instance()` определяет конкретный подкласс `WidgetKit` для создания экземпляра на основании переменной среды, которую определяет `Session`. Аналогичная операция в классе `Session` «выясняет», поддерживаются ли монохромные или цветные дисплеи, и соответственно настраивает конфигурацию одиночного экземпляра `Session`.

Родственные паттерны

С помощью паттерна одиночка могут быть реализованы многие паттерны. См. описания абстрактной фабрики (113), строителя (124) и прототипа (146).

ОБСУЖДЕНИЕ ПОРОЖДАЮЩИХ ПАТТЕРНОВ

Существуют два распространенных способа параметризации системы классами создаваемых ей объектов. Первый способ — порождение подклассов от класса, создающего объекты. Он соответствует паттерну фабричный метод (135). Основной недостаток такого решения — необходимость создания нового подкласса лишь для того, чтобы изменить класс продукта. Это может привести к распространению каскадных изменений. Например, если создатель продукта сам создается фабричным методом, то придется замещать и создателя тоже.

Другой способ параметризации системы в большей степени основан на композиции объектов. Вы определяете объект, которому известно о классах объектов-продуктов, и делаете его параметром системы. Это ключевой аспект таких паттернов, как абстрактная фабрика (113), строитель (124) и прототип (146). Для всех трех характерно создание «фабричного объекта», который изготавливает продукты. В абстрактной фабрике фабричный объект производит объекты разных классов. Фабричный объект строителя постепенно создает сложный продукт, следуя специальному протоколу. Фабричный объект прототипа изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип — это одно и то же, поскольку именно прототип отвечает за возвращение продукта.

Рассмотрим каркас графических редакторов, описанный при обсуждении паттерна прототип. Есть несколько способов параметризовать класс `GraphicTool` классом продукта:

- применить паттерн фабричный метод. Тогда для каждого подкласса класса `Graphic` в палитре будет создан свой подкласс `GraphicTool`. В классе `GraphicTool` будет присутствовать операция `NewGraphic`, переопределяемая каждым подклассом;
- использовать паттерн абстрактная фабрика. Возникнет иерархия классов `GraphicsFactories`, по одной для каждого подкласса `Graphic`. В этом случае каждая фабрика создает только один продукт: `CircleFactory` — окружности `Circle`, `LineFactory` — отрезки `Line` и т. д. `GraphicTool` параметризуется фабрикой для создания подходящих графических объектов;
- применить паттерн прототип. Тогда в каждом подклассе `Graphic` будет реализована операция `Clone`, а `GraphicTool` параметризуется прототипом создаваемого графического объекта.

Выбор оптимального паттерна зависит от многих факторов. В нашем примере каркаса графических редакторов, на первый взгляд, проще всего воспользоваться фабричным методом. Определить новый подкласс `GraphicTool` легко, а экземпляры `GraphicTool` создаются только в момент определения палитры. Основной недостаток такого подхода заключается в комбинаторном росте числа подклассов `GraphicTool`, причем все они почти ничего не делают.

Абстрактная фабрика лишь немногим лучше, поскольку требует создания иерархии классов `GraphicsFactory` такого же размера. Абстрактной фабрике следует отдавать предпочтение перед фабричным методом лишь тогда, когда уже и так существует иерархия класса `GraphicsFactory`: либо потому, что ее автоматически строит компилятор (как в Smalltalk или Objective C), либо она необходима для другой части системы.

В общем, целям каркаса графических редакторов лучше всего отвечает паттерн прототип, поскольку для его применения требуется лишь реализовать операцию `Clone` в каждом классе `Graphics`. Это сокращает число подклассов, а `Clone` можно с пользой применить и для решения других задач — например, для реализации пункта меню `Duplicate` (дублировать), — а не только для создания экземпляров.

С паттерном фабричный метод проект в большей степени поддается настройке при незначительном росте сложности. Другие паттерны нуждаются в создании новых классов, а фабричный метод — только в создании одной новой операции. Часто этот паттерн рассматривается как стандартный способ создания объектов, но вряд ли его стоит рекомендовать в ситуации, когда класс создаваемого экземпляра никогда не изменяется или когда экземпляр

создается внутри операции, которую легко можно заместить в подклассах (например, во время инициализации).

Проекты, в которых используются паттерны абстрактная фабрика, прототип или строитель, оказываются еще более гибкими, чем те, где применяется фабричный метод, но за это приходится платить повышенной сложностью. Часто в начале работы над проектом за основу берется фабричный метод, а позже, когда проектировщик обнаруживает, что решение получается недостаточно гибким, он выбирает другие паттерны. Владение разными паттернами проектирования открывает перед вами широкий выбор при оценке различных критериев.

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.

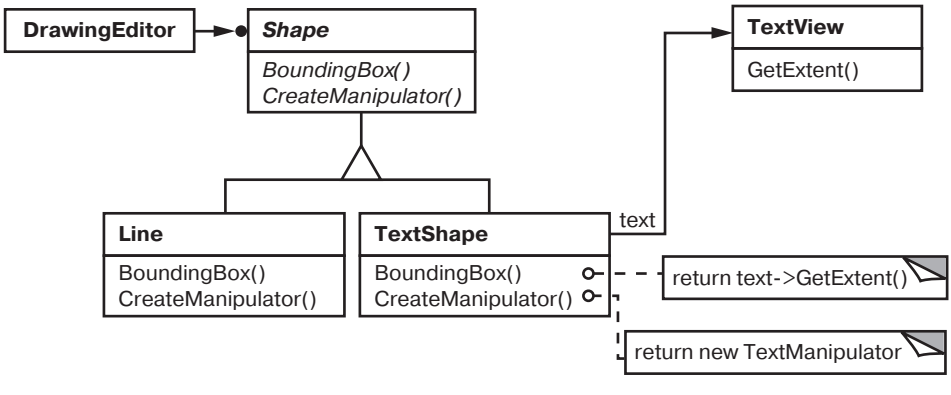
Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т. д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, форма которого может 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.

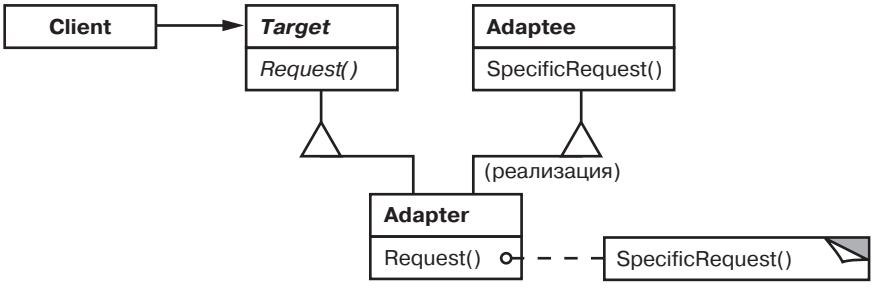
Применимость

Основные условия для применения паттерна адаптер:

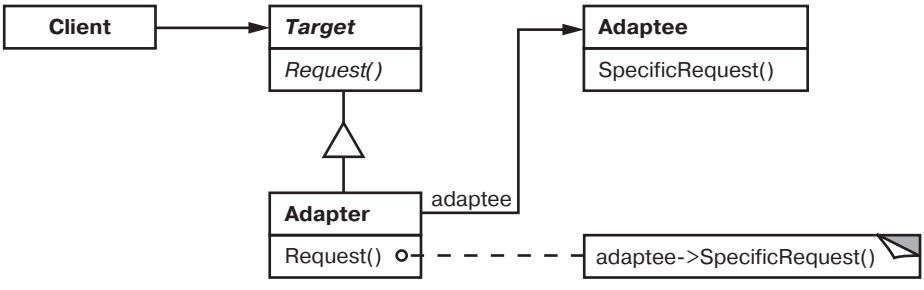
- вы хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- требуется создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (*только для адаптера объектов*!) 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;

- позволяет адаптеру `Adapter` заместить некоторые операции адаптируемого класса `Adaptee`, так как `Adapter` есть не что иное, как подкласс `Adaptee`;
- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру `Adapter` работать со многим адаптируемыми объектами `Adaptee`, то есть с самим `Adaptee` и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- затрудняет замещение операций класса `Adaptee`. Для этого потребуется породить от `Adaptee` подкласс и заставить `Adapter` ссылаться на этот подкласс, а не на сам `Adaptee`.

Ниже перечислены другие аспекты, которые следует рассмотреть, принимая решение о применении паттерна `адаптер`:

- *объем работы по адаптации*. Адаптеры сильно отличаются по объему работы, необходимой для адаптации интерфейса `Adaptee` к интерфейсу `Target`. Это может быть как простейшее преобразование (например, изменение имен операций), так и поддержка совершенно другого набора операций. Объем работы зависит от того, насколько сильно отличаются друг от друга интерфейсы целевого и адаптируемого классов;
- *сменные адаптеры*. Степень повторной используемости класса тем выше, чем меньше предположений делается о тех классах, которые будут его применять. Встраивая адаптацию интерфейса в класс, вы снимаете предположение о том, что другие классы должны «видеть» тот же интерфейс. Другими словами, адаптация интерфейса позволяет включить ваш класс в существующие системы, которые спроектированы для класса с другим интерфейсом. В системе ObjectWorks\Smalltalk [Par90] используется термин *сменный адаптер* (pluggable adapter) для обозначения классов со встроенной адаптацией интерфейса.

  Рассмотрим виджет `TreeDisplay`, позволяющий графически отображать древовидные структуры. Если бы это был специализированный виджет, предназначенный только для одного приложения, то мы могли бы потребовать специального интерфейса от объектов, которые он отображает (например, чтобы все они происходили от абстрактного класса `Tree`). Но если мы хотим сделать его повторно используемым (например, частью библиотеки полезных виджетов), то устанавливать такое требование не-

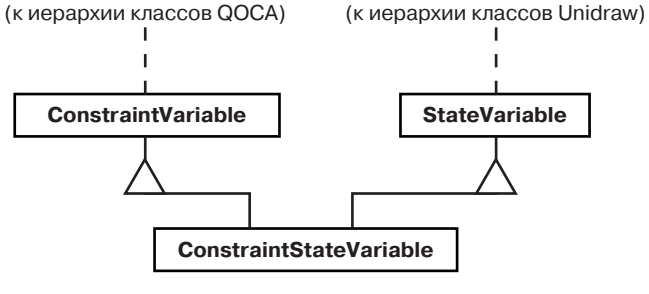
разумно. Скорее всего, разные приложения будут определять собственные классы для представления древовидных структур, и не следует заставлять их пользоваться именно нашим абстрактным классом `Tree`. А у разных структур деревьев будут и разные интерфейсы.

Например, в иерархии каталогов добраться до потомков удастся с помощью операции `GetSubdirectories`, тогда как для иерархии наследования соответствующая операция может называться `GetSubclasses`. Повторно используемый виджет `TreeDisplay` должен «уметь» отображать иерархии обоих видов, даже если у них разные интерфейсы. Другими словами, в `TreeDisplay` должна быть встроена возможность адаптации интерфейсов.

О способах встраивания адаптации интерфейсов в классы говорится в разделе «Реализация»;

- *использование двусторонних адаптеров для обеспечения прозрачности.* Потенциальный недостаток адаптеров заключается в том, что они непрозрачны для всех клиентов. Адаптированный объект уже не обладает интерфейсом `Adaptee`, так что его нельзя использовать там, где `Adaptee` был применим. *Двусторонние адаптеры* способны обеспечить такую прозрачность. А конкретнее, они полезны в тех случаях, когда разные клиенты должны видеть объект по-разному.

  Рассмотрим двусторонний адаптер, который интегрирует каркас графических редакторов Unidraw [VL90] и библиотеку для разрешения ограничений QOCA [HHMV92]. В обеих системах есть классы, явно представляющие переменные: в Unidraw это `StateVariable`, а в QOCA — `ConstraintVariable`. Чтобы заставить Unidraw работать совместно с QOCA, `ConstraintVariable` нужно адаптировать к `StateVariable`. А для того чтобы решения QOCA распространялись на Unidraw, `StateVariable` следует адаптировать к `ConstraintVariable`.



Здесь применен двусторонний адаптер класса `ConstraintStateVariable`, который является подклассом одновременно `StateVariable` и `ConstraintVariable` и адаптирует оба интерфейса друг к другу. Множественное наследование в данном случае вполне приемлемо, поскольку интерфейсы адаптированных классов существенно различаются. Двусторонний адаптер класса соответствует интерфейсам каждого из адаптируемых классов и может работать в любой системе.

Реализация

Хотя реализация адаптера обычно не вызывает затруднений, все же необходимо учитывать ряд аспектов:

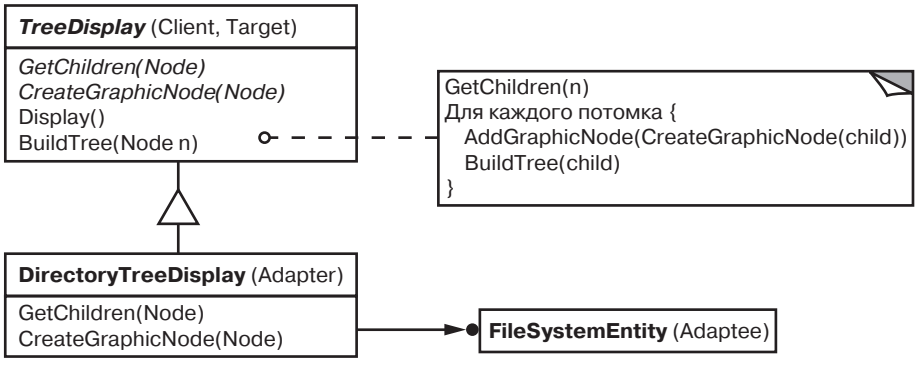
- *реализация адаптеров классов в C++.* В C++ реализация адаптера класса `Adapter` открыто наследует от класса `Target` и закрыто — от `Adaptee`. Таким образом, `Adapter` должен быть подтипом `Target`, но не `Adaptee`;
- *сменные адаптеры.* Рассмотрим три способа реализации сменных адаптеров для описанного выше виджета `TreeDisplay`, который может автоматически отображать иерархические структуры.

  Первый шаг, общий для всех трех реализаций, — поиск «узкого» интерфейса для `Adaptee`, то есть наименьшего подмножества операций, позволяющего выполнить адаптацию. «Узкий» интерфейс, состоящий всего из пары итераций, легче адаптировать, чем интерфейс из нескольких десятков операций. Для `TreeDisplay` адаптации подлежит любая иерархическая структура. Минимальный интерфейс мог бы включать всего две операции: одна определяет графическое представление узла в иерархической структуре, другая — доступ к потомкам узла.

  «Узкий» интерфейс приводит к трем подходам к реализации:

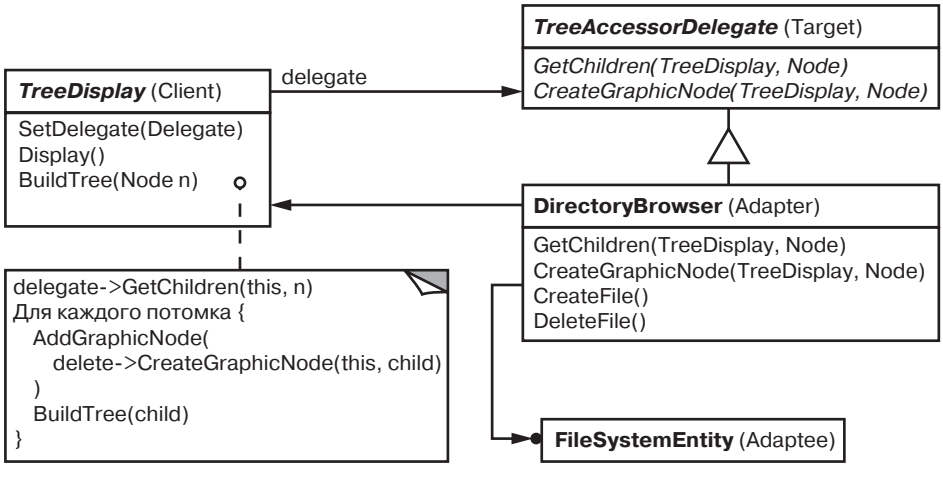
  - *использование абстрактных операций.* Определим в классе `TreeDisplay` абстрактные операции, которые соответствуют «узкому» интерфейсу класса `Adaptee`. Подклассы должны реализовывать эти абстрактные операции и адаптировать иерархически структурированный объект. Например, подкласс `DirectoryTreeDisplay` при их реализации будет осуществлять доступ к структуре каталогов файловой системы.

  `DirectoryTreeDisplay` специализирует узкий интерфейс таким образом, чтобы он мог отображать структуру каталогов, составленную из объектов `FileSystemEntity`;



- *использование объектов-делегатов*. При таком подходе `TreeDisplay` переадресует запросы на доступ к иерархической структуре объекту-делегату. `TreeDisplay` может реализовывать различные стратегии адаптации, подставляя разных делегатов.

Например, предположим, что существует класс `DirectoryBrowser`, который использует `TreeDisplay`. `DirectoryBrowser` может быть делегатом для адаптации `TreeDisplay` к иерархической структуре каталогов. В динамически типизированных языках вроде Smalltalk или Objective C такой подход требует интерфейса для регистрации делегата в адаптере. Тогда `TreeDisplay` просто переадресует запросы делегату. В системе NEXTSTEP [Add94] этот подход активно используется для уменьшения числа подклассов.



В статически типизированных языках вроде C++ требуется явно определять интерфейс для делегата. Специфицировать такой интерфейс можно, поместив «узкий» интерфейс, который необходим классу `TreeDisplay`, в абстрактный класс `TreeAccessorDelegate`. После этого возможно добавить этот интерфейс к выбранному делегату — в данном случае `DirectoryBrowser` — с помощью наследования. Если у `DirectoryBrowser` еще нет существующего родительского класса, то используется одиночное наследование, если есть — множественное. Подобное смешивание классов проще, чем добавление нового подкласса `TreeDisplay` и реализация его операций по отдельности;

- *параметризованные адаптеры*. Обычно в Smalltalk для поддержки сменных адаптеров параметризуют адаптер одним или несколькими блоками. Конструкция блока поддерживает адаптацию без порождения подклассов. Блок может адаптировать запрос, а адаптер может хранить блок для каждого отдельного запроса. В нашем примере это означает, что `TreeDisplay` хранит один блок для преобразования узла в `GraphicNode`, а другой — для доступа к потомкам узла.

Например, чтобы создать класс `TreeDisplay` для отображения иерархии каталогов, мы пишем:

```
directoryDisplay :=
    (TreeDisplay on: treeRoot)
        getChildrenBlock:
            [:node | node getSubdirectories]
        createGraphicNodeBlock:
            [:node | node createGraphicNode].
```

Если интерфейс адаптации встраивается в класс, то этот способ дает удобную альтернативу подклассам.

Пример кода

Приведем краткий обзор реализации адаптеров класса и объекта для примера, обсуждавшегося в разделе «Мотивация», при этом начнем с классов `Shape` и `TextView`:

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
```

```
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Класс `Shape` предполагает, что ограничивающий фигуру прямоугольник определяется двумя противоположными углами. Напротив, в классе `TextView` он определяется начальной точкой, высотой и шириной. В классе `Shape` определена также операция `CreateManipulator` для создания объекта-манипулятора класса `Manipulator`, который знает, как анимировать фигуру в ответ на действия пользователя[1]. В `TextView` эквивалентной операции нет. Класс `TextShape` является адаптером между двумя этими интерфейсами.

Для адаптации интерфейса адаптер класса использует множественное наследование. Принцип адаптера класса состоит в наследовании интерфейса по одной ветви и реализации — по другой. В C++ интерфейс обычно наследуется открыто, а реализация — закрыто. Мы будем придерживаться этого соглашения при определении адаптера `TextShape`:

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

Операция `BoundingBox` преобразует интерфейс `TextView` к интерфейсу `Shape`:

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);
```

[1] CreateManipulator — пример фабричного метода.

```
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

Операция `IsEmpty` демонстрирует прямую переадресацию запросов, общих для обоих классов:

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

Наконец, операция `CreateManipulator` (отсутствующая в классе `TextView`) будет определена с нуля. Предположим, класс `TextManipulator`, который поддерживает манипуляции с `TextShape`, уже реализован:

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Адаптер объектов применяет композицию объектов для объединения классов с разными интерфейсами. При таком решении адаптер `TextShape` содержит указатель на `TextView`:

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

Объект `TextShape` должен инициализировать указатель на экземпляр `TextView`. Делается это в конструкторе. Кроме того, он должен вызывать операции объекта `TextView` всякий раз, как вызываются его собственные операции. В этом примере предполагается, что клиент создает объект `TextView` и передает его конструктору класса `TextShape`:

```
TextShape::TextShape (TextView* t) {
    _text = t;
}
```

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

Реализация `CreateManipulator` не зависит от версии адаптера класса, поскольку реализована с нуля и не использует повторно никакой функциональности `TextView`:

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Сравним этот код с кодом адаптера класса. Для написания адаптера объекта нужно потратить чуть больше усилий, но зато он оказывается более гибким. Например, вариант адаптера объекта `TextShape` будет прекрасно работать и с подклассами `TextView`: клиент просто передает экземпляр подкласса `TextView` конструктору `TextShape`.

Известные применения

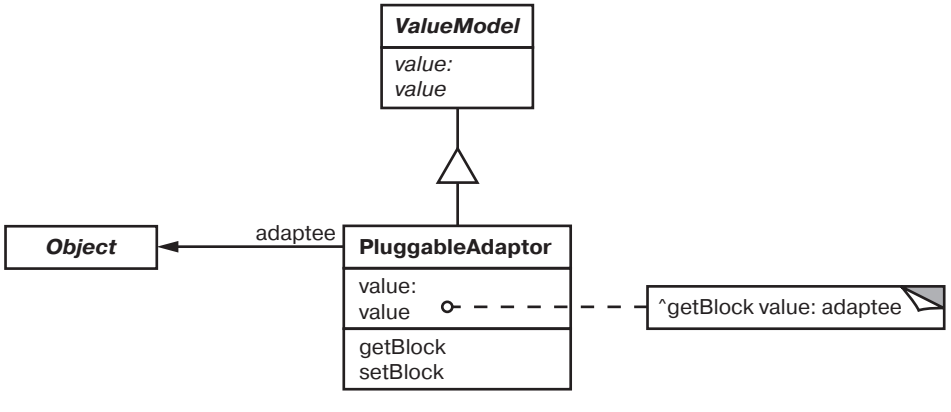
Пример, приведенный в разделе «Мотивация», заимствован из графического приложения ET++Draw, основанного на каркасе ET++ [WGM88]. ET++Draw повторно использует классы ET++ для редактирования текста, применяя для адаптации класс `TextShape`.

В библиотеке InterViews 2.6 определен абстрактный класс `Interactor` для таких элементов пользовательского интерфейса, как полосы прокрутки, кнопки и меню [VL88]. Есть также абстрактный класс `Graphic` для структурированных графических объектов: прямых, окружностей, многоугольников и сплайнов. И `Interactor`, и `Graphic` имеют графическое представление, но у них разные интерфейсы и реализации (общих родительских классов нет), а значит, они несовместимы: нельзя непосредственно вложить структурированный графический объект, скажем, в диалоговое окно.

Вместо этого InterViews 2.6 определяет адаптер объектов `GraphicBlock` — подкласс `Interactor`, который содержит экземпляр `Graphic`. `GraphicBlock` адаптирует интерфейс класса `Graphic` к интерфейсу `Interactor`, позволяет отображать, прокручивать и изменять масштаб экземпляра `Graphic` внутри структуры класса `Interactor`.

Сменные адаптеры широко применяются в системе ObjectWorks\Smalltalk [Par90]. В стандартном Smalltalk определен класс `ValueModel` для представлений, которые отображают единственное значение. Для обращения к значению `ValueModel` определяет интерфейс `value`, `value:`. Эти методы являются абстрактными. Авторы приложений обращаются к значению по имени, более соответствующему предметной области (например, `width` и `width:`), но они не обязаны порождать от `ValueModel` подклассы для адаптации таких зависящих от приложения имен к интерфейсу `ValueModel`.

Вместо этого ObjectWorks\Smalltalk включает подкласс `ValueModel` с именем `PluggableAdaptor`. Объект этого класса адаптирует другие объекты к интерфейсу `ValueModel` (`value, value:`). Его можно параметризовать блоками для получения и установки нужного значения. Внутри `PluggableAdaptor` эти блоки используются для реализации интерфейса `value`, `value:`. Этот класс позволяет также передавать имена селекторов (например, `width, width:`) непосредственно для удобства синтаксиса. Такие селекторы преобразуются в соответствующие блоки автоматически.



Еще один пример из ObjectWorks\Smalltalk — это класс `TableAdaptor`. Он может адаптировать последовательность объектов к табличному представлению. В таблице отображается по одному объекту в строке. Клиент параметризует `TableAdaptor` множеством сообщений, которые используются таблицей для получения от объекта значения в колонках.

В некоторых классах библиотеки NeXT AppKit [Add94] используются объекты-делегаты для реализации интерфейса адаптации. В качестве примера можно привести класс NXBrowser, который способен отображать иерархические списки данных. NXBrowser пользуется объектом-делегатом для обращений и адаптации данных.

Придуманная Скоттом Мейером (Scott Meyer) конструкция «брак по расчету» (Marriage of Convenience) [Mey88] это разновидность адаптера класса. Мейер описывает, как класс `FixedStack` адаптирует реализацию класса `Array` к интерфейсу класса `Stack`. Результат представляет собой стек, содержащий фиксированное число элементов.

Родственные паттерны

Структура паттерна мост (184) аналогична структуре адаптера, но у моста иное назначение. Он отделяет интерфейс от реализации, чтобы то и другое можно было изменять независимо. Адаптер же призван изменить интерфейс *существующего* объекта.

Паттерн декоратор (209) расширяет функциональность объекта, изменяя его интерфейс. Таким образом, декоратор более прозрачен для приложения, чем адаптер. Как следствие, декоратор поддерживает рекурсивную композицию, что для «чистых» адаптеров невозможно.

Заместитель (246) определяет представителя или суррогат другого объекта, но не изменяет его интерфейс.

ПАТТЕРН BRIDGE (МОСТ)

Название и классификация паттерна

Мост — паттерн, структурирующий объекты.

Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Другие названия

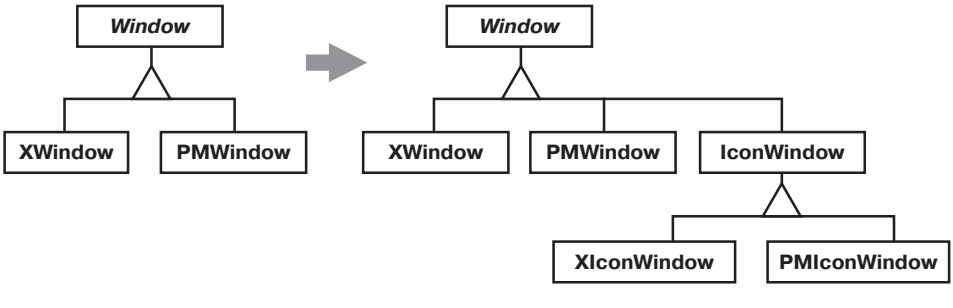
`Handle/Body` (описатель/тело).

Мотивация

Если некоторая абстракция может иметь несколько возможных реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализации.

Рассмотрим реализацию переносимой абстракции окна в библиотеке для разработки пользовательских интерфейсов. Написанные с ее помощью приложения должны работать в разных средах, например под X Window System и Presentation Manager (PM) от компании IBM. С помощью наследования можно было бы определить абстрактный класс `Window` и его подклассы `XWindow` и `PMWindow`, реализующие интерфейс окна для разных платформ. Но у такого решения есть два недостатка:

- абстракцию `Window` неудобно расширять для новых видов окон или новых платформ. Представьте себе подкласс `IconWindow`, который специализирует абстракцию окна для пиктограмм. Чтобы поддержать пиктограммы на обеих платформах, нам придется реализовать два новых подкласса `XIconWindow` и `PMIconWindow`. Более того, по два подкласса необходимо определять для *каждого* вида окон. А для поддержки третьей платформы придется определять для всех видов окон новый подкласс `Window`;

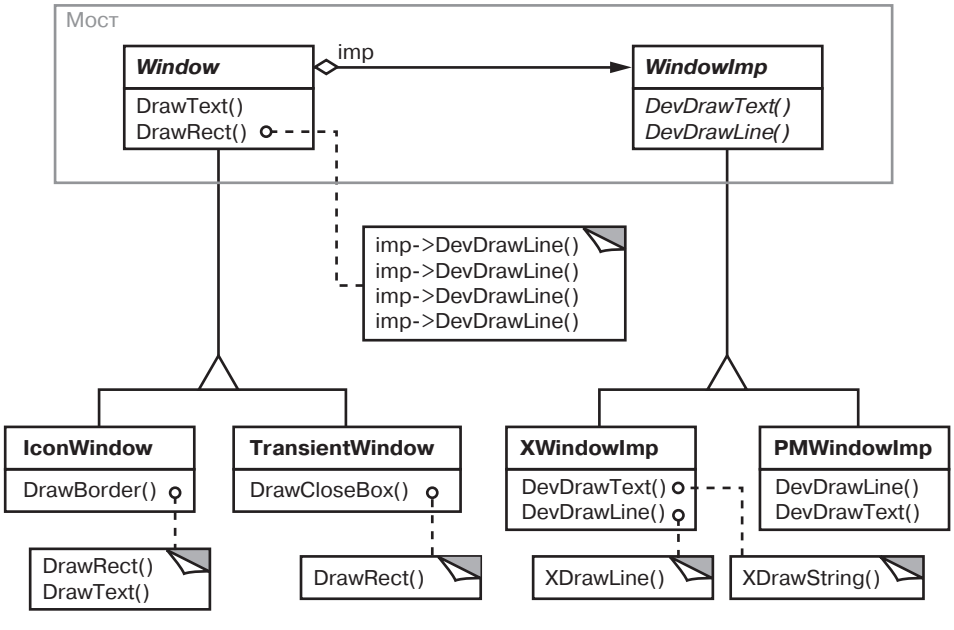


- клиентский код становится платформеннозависимым. При создании окна клиент создает экземпляр конкретного класса, имеющего вполне определенную реализацию. Например, создавая объект `XWindow`, мы привязываем абстракцию окна к ее реализации для системы X Window; следовательно, код клиента становится ориентированным именно на

эту оконную систему. В свою очередь, это усложняет перенос клиента на другие платформы.

Клиенты должны иметь возможность создавать окно без привязки к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение. Поэтому в клиентском коде экземпляры окон должны создаваться без упоминания конкретных платформ.

Паттерн мост решает все эти проблемы: абстракция окна и ее реализация помещаются в раздельные иерархии классов. Таким образом, существует одна иерархия для интерфейсов окон (`Window`, `IconWindow`, `TransientWindow`) и другая (с корнем `WindowImp`) — для платформеннозависимых реализаций. Так, подкласс `XWindowImp` предоставляет реализацию для системы X Window System.



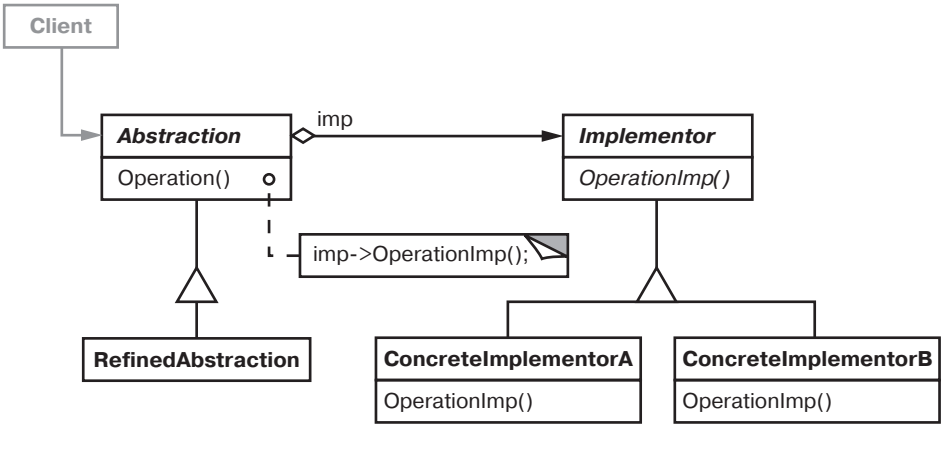
Все операции подклассов `Window` реализованы в категориях абстрактных операций из интерфейса `WindowImp`. Это отделяет абстракцию окна от различных ее платформенно-зависимых реализаций. Отношение между классами `Window` и `WindowImp` мы будем называть *мостом*, поскольку между абстракцией и реализацией строится мост, и они могут изменяться независимо.

Применимость

Основные условия для применения паттерна мост:

- *требуется избежать постоянной привязки абстракции к реализации*. Так, например, бывает, когда реализация должна выбираться во время выполнения программы;
- *и абстракции, и реализации должны расширяться новыми подклассами*. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- *изменения в реализации абстракции не должны отражаться на клиентах*, то есть клиентский код не должен перекомпилироваться;
- *(только для C++) требуется полностью скрыть от клиентов реализацию абстракции*. В C++ представление класса видимо через его интерфейс;
- *число классов стремительно разрастается*, как на первой диаграмме из раздела «Мотивация». Это признак того, что иерархию следует разделить на две части. Для таких иерархий классов Рамбо (Rumbaugh) использует термин «вложенные обобщения» [RBP+91];
- *реализация должна совместно использоваться несколькими объектами* (например, на базе подсчета ссылок), и этот факт должен быть скрыт от клиента. Простой пример — это разработанный Джеймсом Коплиеном класс `String` [Cop92], в котором разные объекты могут разделять одно и то же представление строки (`StringRep`).

Структура



Участники

- **Abstraction** (`Window`) — абстракция:
  - определяет интерфейс абстракции;
  - хранит ссылку на объект типа `Implementor`;
- **RefinedAbstraction** (`IconWindow`) — уточненная абстракция:
  - расширяет интерфейс, определенный абстракцией `Abstraction`;
- **Implementor** (`WindowImp`) — реализатор:
  - определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса `Abstraction`. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса `Implementor` предоставляет только примитивные операции, а класс `Abstraction` определяет операции более высокого уровня, основанные на этих примитивах;
- **ConcreteImplementor** (`XWindowImp`, `PMWindowImp`) — конкретный реализатор:
  - реализует интерфейс класса `Implementor` и определяет его конкретную реализацию.

Отношения

Объект `Abstraction` перенаправляет запросы клиента своему объекту `Implementor`.

Результаты

Результаты применения паттерна мост:

- *отделение реализации от интерфейса*. Реализация больше не имеет постоянной привязки к интерфейсу. Реализация абстракции может настраиваться во время выполнения. Объект может даже динамически изменять свою реализацию.

  Разделение классов `Abstraction` и `Implementor` устраняет также зависимости от реализации, устанавливаемые на этапе компиляции. Чтобы изменить класс реализации, не обязательно перекомпилировать класс `Abstraction` и его клиентов. Это свойство особенно важно, если необходимо обеспечить двоичную совместимость между разными версиями библиотеки классов.

  Кроме того, такое разделение облегчает разбиение системы на слои и тем самым позволяет улучшить ее структуру. Высокоуровневые части системы должны знать только о классах `Abstraction` и `Implementor`;

- *повышение степени расширяемости.* Иерархии классов `Abstraction` и `Implementor` могут расширяться независимо;
- *сокрытие деталей реализации от клиентов.* Клиентов можно изолировать от таких подробностей реализации, как совместное использование объектов класса `Implementor` и сопутствующего механизма подсчета ссылок.

Реализация

Если вы намереваетесь применить паттерн мост, то подумайте о таких аспектах реализации:

- *только один класс Implementor.* В ситуациях, когда есть только одна реализация, создавать абстрактный класс `Implementor` необязательно. Это вырожденный случай паттерна мост — между классами `Abstraction` и `Implementor` существует взаимно однозначное соответствие. Тем не менее разделение все же полезно, если изменение реализации класса не должно отражаться на существующих клиентах (должно быть достаточно заново скомпоновать программу, не перекомпилируя клиентский код).

  Для описания такого разделения Каролан (Carolan) [Car89] употребляет сочетание «чеширский кот». В C++ интерфейс класса `Implementor` можно определить в закрытом заголовочном файле, который не передается клиентам. Это позволяет полностью скрыть реализацию класса от клиентов;

- *создание правильного объекта Implementor.* Как, когда и где принимается решение о том, экземпляр какого из классов `Implementor` следует создать, если таких классов несколько?

  Если класс `Abstraction` располагает полной информацией обо всех классах `ConcreteImplementor`, то он может создать один из них в своем конструкторе; какой именно — зависит от переданных конструктору параметров. Так, если класс коллекции поддерживает несколько реализаций, то решение может приниматься в зависимости от размера коллекции. Для небольших коллекций применяется реализация в виде связанного списка, для больших — в виде хеш-таблиц.

  Другой подход — заранее выбрать реализацию по умолчанию, а позже изменять ее в соответствии с тем, как она используется. Например, если число элементов в коллекции превышает некоторую условную величину, то мы переключаемся с одной реализации на другую, более эффективную.

Также можно делегировать решение другому объекту. В примере с иерархиями `Window/WindowImp` уместно было бы ввести фабричный объект (см. паттерн абстрактная фабрика (113)), единственная задача которого — инкапсулировать платформенную специфику. Фабрика обладает информацией, объекты `WindowImp` какого вида надо создавать для данной платформы, а объект `Window` просто обращается к ней с запросом о предоставлении какого-нибудь объекта `WindowImp` и получает то, что нужно. Преимущество описанного подхода: класс `Abstraction` напрямую не привязан ни к одному из классов `Implementor`;

- *совместное использование реализаторов*. Джеймс Коплиен показал, как в C++ можно применить идиому «описатель/тело», чтобы несколькими объектами могла совместно использоваться одна и та же реализация [Cop92]. В теле хранится счетчик ссылок, который увеличивается и уменьшается в классе описателя. Код для присваивания значений описателям, совместно использующим одно тело, в общем виде выглядит так:

```
Handle& Handle::operator= (const Handle& other)  {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

- *применение множественного наследования*. В C++ для объединения интерфейса с его реализацией можно воспользоваться множественным наследованием [Mar91]. Например, класс может открыто наследовать классу `Abstraction` и закрыто — классу `ConcreteImplementor`. Но такое решение зависит от статического наследования и жестко привязывает реализацию к ее интерфейсу. Поэтому реализовать настоящий мост с помощью множественного наследования невозможно — по крайней мере в C++.

Пример кода

В следующем коде на C++ реализован пример `Window/WindowImp`, который обсуждался в разделе «Мотивация». Класс `Window` определяет абстракцию окна для клиентских приложений:

```
class Window {
public:
    Window(View* contents);

    // запросы, обрабатываемые окном
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // запросы, перенаправляемые реализации
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // содержимое окна
};
```

В классе `Window` хранится ссылка на `WindowImp` — абстрактный класс, в котором объявлен интерфейс к данной оконной системе:

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // множество других функций для рисования в окне...
protected:
    WindowImp();
};
```

Подклассы `Window` определяют различные виды окон, как то: окно приложения, пиктограмма, временное диалоговое окно, плавающая палитра инструментов и т. д.

Например, класс `ApplicationWindow` реализует операцию `DrawContents` для отрисовки содержимого экземпляра класса `View`, который в нем хранится:

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

А в классе `IconWindow` хранится имя растрового изображения для пиктограммы:

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

и реализация операции `DrawContents` для рисования этого изображения в окне:

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

Существует много других разновидностей класса `Window`. Окну класса `TransientWindow` иногда необходимо как-то сообщаться с создавшим его окном во время диалога, поэтому в объекте класса хранится ссылка на создателя. Окно класса `PaletteWindow` всегда располагается поверх других. Окно класса `IconDockWindow` (контейнер пиктограмм) хранит окна класса `IconWindow` и размещает их в ряд.

Операции класса `Window` определяются в категориях интерфейса `WindowImp`. Например, `DrawRect` вычисляет координаты по двум своим параметрам `Point`, перед тем как вызвать операцию `WindowImp`, которая рисует в окне прямоугольник:

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Конкретные подклассы `WindowImp` поддерживают разные оконные системы. Так, класс `XWindowImp` ориентирован на систему X Window:

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // прочие операции открытого интерфейса...
private:
    // переменные, описывающие состояние, специфическое
    // для X Window system:
    Display* _dpy;
    Drawable _winid;  // идентификатор окна
    GC _gc;           // графический контекст окна
};
```

Для Presentation Manager (PM) определяется класс `PMWindowImp`:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // прочие операции открытого интерфейса...
private:
    // переменные, описывающие состояние, специфическое для PM:
    HPS _hps;
};
```

Эти подклассы реализуют операции `WindowImp` в контексте примитивов оконной системы. Например, `DeviceRect` для X Window реализуется так:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
```

```
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

А реализация для PM выглядит так:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR)  ||
        (GpiEndPath(_hps) == false)
    ) {
        // сообщить об ошибке

    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Как окно получает экземпляр нужного подкласса `WindowImp`? В данном примере предполагается, что за это отвечает класс `Window`. Его операция `GetWindowImp` получает подходящий экземпляр от абстрактной фабрики (см. описание паттерна абстрактная фабрика (113)), которая инкапсулирует все зависимости от оконной системы.

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

`WindowSystemFactory::Instance()` возвращает абстрактную фабрику, которая изготавливает все системнозависимые объекты. Для простоты мы сделали эту фабрику одиночкой (157) и позволили классу `Window` обращаться к ней напрямую.

Известные применения

Пример класса `Window` позаимствован из ET++ [WGM88]. В ET++ класс `WindowImp` называется `WindowPort` и имеет такие подклассы, как `XWindowPort` и `SunWindowPort`. Объект `Window` создает соответствующего себе реализатора `Implementor`, запрашивая его у абстрактной фабрики, которая называется `WindowSystem`. Эта фабрика предоставляет интерфейс для создания платформеннозависимых объектов: шрифтов, курсоров, растровых изображений и т. д.

Дизайн классов `Window`/`WindowPort` в ET++ обобщает паттерн мост в том отношении, что `WindowPort` сохраняет также обратную ссылку на `Window`. Класс-реализатор `WindowPort` использует эту ссылку для уведомления `Window` о событиях, специфичных для `WindowPort`: поступлении событий ввода, изменениях размера окна и т. д.

В работах Джеймса Коплиена [Cop92] и Бьерна Страуструпа [Str91] упоминаются классы описателей и приводятся некоторые примеры. Основной акцент в этих примерах сделан на аспектах управления памятью, например совместном использовании представления строк и поддержки объектов переменного размера. Нас же в первую очередь интересует поддержка независимых расширений абстракции и ее реализации.

В библиотеке libg++ [Lea88] определены классы, которые реализуют распространенные структуры данных: `Set` (множество), `LinkedSet` (множество как связанный список), `HashSet` (множество как хеш-таблица), `LinkedList` (связанный список) и `HashTable` (хеш-таблица). `Set` — это абстрактный класс, определяющий абстракцию множества, а `LinkedList` и `HashTable` — конкретные реализации связанного списка и хеш-таблицы. `LinkedSet` и `HashSet` — реализаторы абстракции `Set`, формирующие мост между `Set` и `LinkedList` и `HashTable` соответственно. Перед вами пример вырожденного моста, поскольку абстрактного класса `Implementor` здесь нет.

В библиотеке NeXT AppKit [Add94] паттерн мост используется при реализации и отображении графических изображений. Рисунок может быть представлен по-разному. Оптимальный способ его отображения на экране зависит от свойств дисплея и прежде всего от числа цветов и разрешения. Если бы не AppKit, то для каждого приложения разработчикам пришлось

бы самостоятельно выяснять, какой реализацией пользоваться в конкретных условиях.

Чтобы избавить разработчика от этой ответственности, AppKit предоставляет мост `NXImage`/`NXImageRep`. Класс `NXImage` определяет интерфейс для обработки изображений. Реализация же определена в отдельной иерархии классов `NXImageRep`, в которой есть такие подклассы, как `NXEPSImageRep`, `NXCachedImageRep` и `NXBitMapImageRep`. В классе `NXImage` хранятся ссылки на один или более объектов `NXImageRep`. Если имеется более одной реализации изображения, то `NXImage` выбирает самую подходящую для данного дисплея. При необходимости `NXImage` даже может преобразовать изображение из одного формата в другой. Интересная особенность этого варианта моста в том, что `NXImage` может одновременно хранить несколько реализаций `NXImageRep`.

Родственные паттерны

Паттерн абстрактная фабрика (113) может создать и сконфигурировать мост.

Для обеспечения совместной работы не связанных между собой классов прежде всего предназначен паттерн адаптер (146). Обычно он применяется в уже готовых системах. Мост же участвует в проекте с самого начала и призван поддержать возможность независимого изменения абстракций и их реализаций.

ПАТТЕРН COMPOSITE (КОМПОНОВЩИК)

Название и классификация паттерна

Компоновщик — паттерн, структурирующий объекты.

Назначение

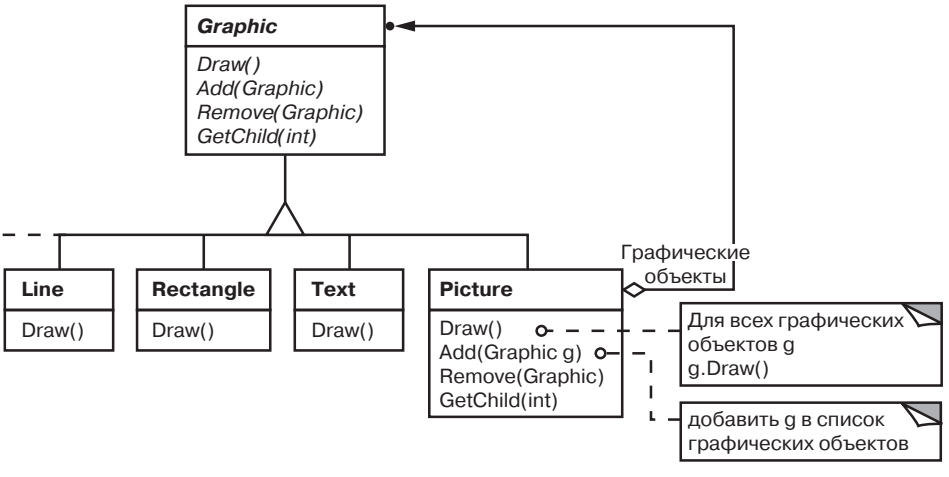
Компонует объекты в древовидные структуры для представления иерархий «часть — целое». Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Мотивация

Такие приложения, как графические редакторы и редакторы электрических схем, позволяют пользователям строить сложные схемы из более простых компонентов. Проектировщик может сгруппировать мелкие компоненты для формирования более крупных, которые, в свою очередь, могут стать ос-

новой для создания еще более крупных. В простой реализации можно было бы определить классы графических примитивов, например текста и линий, а также классы, используемые в качестве контейнеров для этих примитивов.

Но у такого решения есть существенный недостаток. Программа, в которой эти классы используются, должна по-разному обращаться с примитивами и контейнерами, хотя пользователь чаще всего работает с ними единообразно. Необходимость различать эти объекты усложняет приложение. Паттерн компоновщик описывает, как можно применить рекурсивную композицию таким образом, что клиенту не придется проводить различие между простыми и составными объектами.



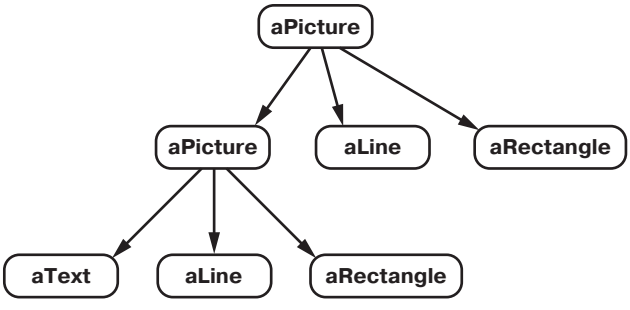
Ключом к паттерну компоновщик является абстрактный класс, который представляет *одновременно* и примитивы, и контейнеры. В графической системе этот класс может называться `Graphic`. В нем объявляются операции, специфичные для каждого вида графического объекта (такие как `Draw`), а также общие для всех составных объектов, например операции для доступа и управления потомками.

Подклассы `Line`, `Rectangle` и `Text` (см. схему выше) определяют примитивные графические объекты. В них операция `Draw` реализована соответственно для рисования прямых, прямоугольников и текста. Поскольку у примитивных объектов нет потомков, то ни один из этих подклассов не реализует операции, относящиеся к управлению потомками.

Класс `Picture` определяет агрегат, состоящий из объектов `Graphic`. Реализованная в нем операция `Draw` вызывает одноименную функцию для каждого

потомка, а операции для работы с потомками уже не пусты. Поскольку интерфейс класса `Picture` соответствует интерфейсу `Graphic`, то в состав объекта `Picture` могут входить и другие такие же объекты.

На следующей схеме показана типичная структура составного объекта, рекурсивно скомпонованного из объектов класса `Graphic`.

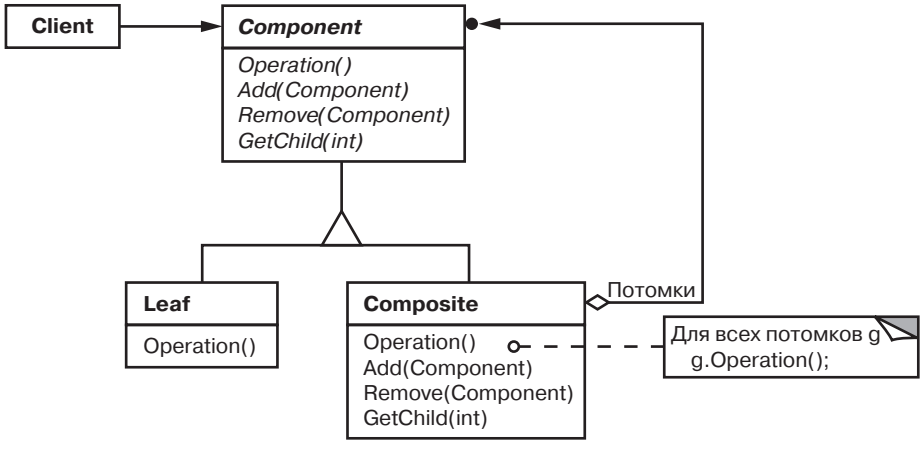


Применимость

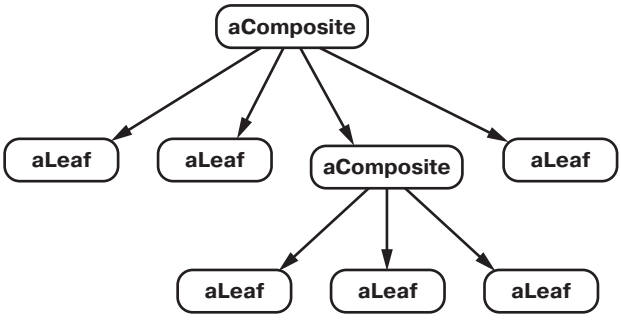
Основные условия для применения паттерна компоновщик:

- требуется представить иерархию объектов вида «часть — целое»;
- клиенты должны по единым правилам работать с составными и индивидуальными объектами.

Структура



Структура типичного составного объекта могла бы выглядеть так:



Участники

- **Component** (`Graphic`) — компонент:
  - объявляет интерфейс для компонуемых объектов;
  - реализует поведение по умолчанию для интерфейсов, общих для всех классов;
  - объявляет интерфейс для обращения к потомкам и управления ими;
  - (не обязательно) определяет интерфейс для обращения к родителю компонента в рекурсивной структуре и при необходимости реализует его;
- **Leaf** (`Rectangle`, `Line`, `Text` и т. п.) — лист:
  - представляет листовые узлы композиции и не имеет потомков;
  - определяет поведение примитивных объектов в композиции;
- **Composite** (`Picture`) — составной объект:
  - определяет поведение компонентов, у которых есть потомки;
  - хранит компоненты-потомки;
  - реализует относящиеся к управлению потомками операции в интерфейсе класса `Component`;
- **Client** — клиент:
  - манипулирует объектами композиции через интерфейс `Component`.

Отношения

Клиенты используют интерфейс класса `Component` для взаимодействия с объектами в составной структуре. Если получателем запроса является

листовый объект `Leaf`, то он и обрабатывает запрос. Когда же получателем является составной объект `Composite`, то обычно он перенаправляет запрос своим потомкам — возможно, с выполнением некоторых дополнительных операций до или после перенаправления.

Результаты

Паттерн компоновщик:

- *определяет иерархии классов, состоящие из примитивных и составных объектов*. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий получить примитивный объект, может работать и с составным;
- *упрощает архитектуру клиента*. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- *облегчает добавление новых видов компонентов*. Новые подклассы классов `Composite` или `Leaf` будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиент при добавлении новых компонентов не нужно;
- *способствует созданию общего дизайна*. Впрочем, такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно установить ограничения на то, какие объекты могут входить в состав композиции. Иногда бывает нужно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого приходится проводить проверки во время выполнения.

Реализация

При реализации паттерна компоновщик приходится учитывать целый ряд аспектов:

- *явные ссылки на родителей*. Хранение в компоненте ссылки на своего родителя может упростить обход структуры и управление ею. Наличие такой ссылки облегчает передвижение вверх по структуре и удаление

компонента. Кроме того, ссылки на родителей помогают реализовать паттерн цепочка обязанностей (263).

Обычно ссылку на родителя определяют в классе `Component`. Классы `Leaf` и `Composite` могут наследовать саму ссылку и операции с ней.

При наличии ссылки на родителя важно поддерживать следующий инвариант: для всех потомков в составном объекте родителем является составной объект, для которого они в свою очередь являются потомками. Простейший способ гарантировать соблюдение этого условия — изменять родителя компонента *только* тогда, когда он добавляется или удаляется из составного объекта. Если это удается один раз реализовать в операциях `Add` и `Remove`, то реализация будет унаследована всеми подклассами, а следовательно, инвариант будет поддерживаться автоматически;

- *совместное использование компонентов.* Часто бывает полезно организовать совместное использование компонентов — например, для уменьшения объема занимаемой памяти. Но если у компонента может быть более одного родителя, то совместное использование становится проблемой.

  Возможное решение — позволить компонентам хранить ссылки на нескольких родителей. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности. Паттерн приспособленец (231) показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения родителей. Работает он в тех случаях, когда потомки могут избежать отправки сообщений своим родителям, вынеся за свои границы часть внутреннего состояния;

- *максимизация интерфейса класса Component.* Одна из целей паттерна компоновщик — избавить клиентов от необходимости знать, работают ли они с листовым или составным объектом. Для достижения этой цели класс `Component` должен сделать как можно больше операций общими для классов `Composite` и `Leaf`. Обычно класс `Component` предоставляет для этих операций реализации по умолчанию, а подклассы `Composite` и `Leaf` замещают их.

  Однако иногда эта цель вступает в конфликт с принципом проектирования иерархии классов, согласно которому класс должен определять только логичные для всех его подклассов операции. Класс `Component` поддерживает много операций, не имеющих смысла для класса `Leaf`. Как же тогда предоставить для них реализацию по умолчанию?

Иногда некоторая изобретательность позволяет перенести в класс `Component` операцию, которая, на первый взгляд, имеет смысл только для составных объектов. Например, интерфейс для обращений к потомкам является фундаментальной частью класса `Composite`, но вовсе не обязательно класса `Leaf`. Однако если рассматривать `Leaf` как `Component`, у которого *никогда* не бывает потомков, то в классе `Component` можно определить операцию обращения к потомкам как никогда не возвращающую потомков. Тогда подклассы `Leaf` могут использовать эту реализацию по умолчанию, а в подклассах `Composite` она будет переопределена, чтобы возвращать потомков.

Операции управления потомками создают немало проблем; они будут рассмотрены в следующем разделе;

- *объявление операций для управления потомками*. Хотя в классе `Composite` реализованы операции Add и Remove для добавления и удаления потомков, но для паттерна компоновщик важно, в каких классах эти операции *объявлены*. Надо ли объявлять их в классе `Component` и тем самым делать доступными в `Leaf`, или их следует объявить и определить только в классе `Composite` и его подклассах?

  Ответ на этот вопрос подразумевает компромисс между безопасностью и прозрачностью:

  - если определить интерфейс для управления потомками в корне иерархии классов, мы добиваемся прозрачности, так как все компоненты удается трактовать единообразно. Однако за это расплачиваться приходится безопасностью, поскольку клиент может попытаться выполнить бессмысленное действие вроде добавления или удаления объекта из листового узла;
  - если управление потомками определяется в классе `Composite`, то безопасность будет обеспечена — ведь любая попытка добавить или удалить объекты из листьев в статически типизированном языке вроде C++ будет перехвачена на этапе компиляции. Но прозрачность при этом теряется, так как листовые и составные объекты обладают разными интерфейсами.

  В паттерне компоновщик мы придаем особое значение прозрачности, а не безопасности. Если для вас важнее безопасность, будьте готовы к тому, что в некоторых случаях вы можете потерять информацию о типе, и компонент придется преобразовывать к типу составного объекта. Как это сделать, не прибегая к небезопасным приведениям типов?

Можно, например, объявить в классе `Component` операцию `Composite* GetComposite()`. Класс `Component` реализует ее по умолчанию, возвращая `null`-указатель. А в классе `Composite` эта операция переопределена, чтобы она возвращала текущий объект в виде указателя `this`:

```
class Composite;

class Component {
public:
    // ...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

Благодаря операции `GetComposite` можно спросить у компонента, является ли он составным. К возвращаемому этой операцией составному объекту допустимо безопасно применять операции `Add` и `Remove`:

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf); // лист не добавляется
}
```

Аналогичные проверки на принадлежность классу `Composite` в C++ выполняются с помощью оператора `dynamic_cast`.

Разумеется, недостаток такого подхода заключается в том, что мы не обращаемся со всеми компонентами единообразно. Снова приходится проверять тип, перед тем как предпринять то или иное действие.

Единственный способ обеспечить прозрачность — это включить в класс `Component` реализации операций `Add` и `Remove` по умолчанию. Но тогда появится новая проблема: нельзя реализовать `Component::Add` так, чтобы не появилась возможность ошибки. Можно, конечно, сделать данную операцию пустой, но тогда нарушается важное проектное ограничение: попытка добавить что-то в листовый объект, скорее всего, свидетельствует об ошибке. Допустимо было бы заставить ее удалять свой аргумент, но такое поведение может оказаться неожиданным для клиента.

Обычно лучшим решением является такая реализация `Add` и `Remove` по умолчанию, при которой они завершаются с ошибкой (возможно, возбуждая исключение), если компоненту не разрешено иметь потомков (для `Add`) или аргумент не является чьим-либо потомком (для `Remove`).

Другая возможность — слегка изменить семантику операции «удаление». Если компонент хранит ссылку на родителя, то можно было бы считать, что `Component::Remove` удаляет самого себя. Тем не менее, для операции `Add` по-прежнему нет разумной интерпретации;

- *должен ли Component реализовывать список компонентов*? Возможно, вам захочется определить множество потомков в виде переменной экземпляра класса `Component`, в котором объявлены операции обращения к потомкам и управления ими. Но размещение указателя на потомков в базовом классе создает лишние затраты памяти во всех листовых узлах, хотя у листа потомков быть не может. Такое решение может использоваться только в том случае, если в структуре относительно мало потомков;
- *упорядочение потомков*. Во многих случаях важен порядок следования потомков составного объекта. В рассмотренном выше примере класса `Graphic` под порядком может пониматься Z-порядок расположения потомков. В составных объектах, описывающих деревья синтаксического разбора, составные операторы могут быть экземплярами класса `Composite`, порядок следования потомков которых отражает семантику программы.

  Если порядок следования потомков важен, необходимо учитывать его при проектировании интерфейсов доступа и управления потомками. В этом может помочь паттерн итератор (302);

- *кэширование для повышения производительности.* Если приходится часто выполнять обход или поиск в композициях, то класс `Composite` может кэшировать либо непосредственно полученные результаты, либо только информацию, достаточную для ускорения обхода или поиска. Например, класс `Picture` из примера, приведенного в разделе «Мотивация», мог бы кэшировать охватывающие прямоугольники своих потомков. При рисовании или выделении эта информация позволила бы пропускать тех потомков, которые не видимы в текущем окне.

  При любом изменении компонента кэшированная информация всех его родителей должна становиться недействительной. Наиболее эффективен такой подход в случае, когда компонентам известно об их родителях. Поэтому, если вы решите воспользоваться кэшированием, необходимо определить интерфейс, позволяющий уведомить составные объекты о недействительности содержимого их кэшей;

- *кто должен удалять компоненты?* В языках, где нет уборщика мусора, лучше всего поручить классу `Composite` удалять своих потомков в момент уничтожения. Исключением из этого правила является случай, когда листовые объекты постоянны и, следовательно, могут использоваться совместно;

- *выбор структуры данных для хранения компонентов.* Составные объекты могут хранить своих потомков в самых разных структурах данных, включая связанные списки, деревья, массивы и хеш-таблицы. Выбор структуры данных определяется, как всегда, эффективностью. Собственно говоря, вовсе не обязательно пользоваться какой-либо из универсальных структур. Иногда в составных объектах каждый потомок представляется отдельной переменной. Правда, для этого каждый подкласс `Composite` должен реализовывать свой собственный интерфейс управления памятью. См. пример в описании паттерна интерпретатор.

Пример кода

Такие изделия, как компьютеры и стереосистемы, часто имеют иерархическую структуру. Например, в раме монтируются дисковые накопители и плоские электронные платы, к шине подсоединяются различные карты, а корпус содержит раму, шины и т. д. Подобные структуры моделируются с помощью паттерна компоновщик.

Класс `Equipment` определяет интерфейс для всех видов аппаратуры в иерархии вида «часть — целое»:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

В классе `Equipment` объявлены операции, которые возвращают атрибуты аппаратного блока, например энергопотребление и стоимость. Подклассы реализуют эти операции для конкретных видов оборудования. Класс `Equipment` объявляет также операцию `CreateIterator`, возвращающую итератор `Iterator` (см. приложение В) для обращения к отдельным частям. Реализация этой операции по умолчанию возвращает итератор `NullIterator`, умеющий обходить только пустое множество.

Среди подклассов `Equipment` могут быть листовые классы, представляющие дисковые накопители, микросхемы и переключатели:

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

`CompositeEquipment` — это базовый класс для оборудования, содержащего другое оборудование. Одновременно это подкласс класса `Equipment`:

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
```

```
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
```

`CompositeEquipment` определяет операции для доступа и управления внутренними аппаратными блоками. Операции `Add` и `Remove` добавляют и удаляют оборудование из списка, хранящегося в переменной `_equipment`. Операция `CreateIterator` возвращает итератор (точнее, экземпляр класса `ListIterator`), который будет обходить этот список.

Реализация по умолчанию операции `NetPrice` могла бы использовать `CreateIterator` для суммирования цен на отдельные блоки[1]:

```
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

Теперь мы можем представить аппаратный блок компьютера в виде подкласса к `CompositeEquipment` под названием `Chassis`. `Chassis` наследует порожденные операции класса `CompositeEquipment`.

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

[1] Очень легко забыть об удалении итератора после завершения работы с ним. В описании паттерна итератор рассказано, как защититься от таких ошибок.

Аналогичным образом можно определить и другие контейнеры для оборудования, например `Cabinet` (корпус) и `Bus` (шина). Этого вполне достаточно для сборки из отдельных блоков (довольно простого) персонального компьютера:

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

Известные применения

Примеры паттерна компоновщик встречаются почти во всех объектно-ориентированных системах. Первоначально класс `View` в схеме модель/представление/контроллер в языке Smalltalk [KP88] был компоновщиком, и почти все библиотеки для построения пользовательских интерфейсов и каркасы проектировались аналогично. Среди них ET++ (со своей библиотекой VObjects [WGM88]) и InterViews (классы Styles [LCI+92], Graphics [VL88] и Glyphs [CL90].) Интересно отметить, что первоначально вид `View` имел несколько подвидов, то есть он был одновременно и классом `Component`, и классом `Composite`. В версии 4.0 языка Smalltalk-80 схема модель/представление/контроллер была пересмотрена, в нее ввели класс `VisualComponent`, подклассами которого являлись `View` и `CompositeView`.

В каркасе для построения компиляторов RTL, который написан на Smalltalk [JML92], паттерн компоновщик используется очень широко. `RTLExpression` — это разновидность класса `Component` для построения деревьев синтаксического разбора. У него есть подклассы, например `BinaryExpression`, потомками которого являются объекты класса `RTLExpression`. В совокупности эти классы определяют составную структуру для деревьев разбора. `RegisterTransfer` — класс `Component` для промежуточной формы представления программы SSA (Single Static Assignment). Листовые подклассы `RegisterTransfer` определяют различные статические присваивания, например:

- примитивные присваивания, которые выполняют операцию над двумя регистрами и сохраняют результат в третьем;

- присваивание, у которого есть исходный, но нет целевого регистра. Следовательно, регистр используется после возврата из процедуры;
- присваивание, у которого есть целевой, но нет исходного регистра. Это означает, что присваивание регистру происходит перед началом процедуры.

Подкласс `RegisterTransferSet` является примером класса `Composite` для представления присваиваний, изменяющих сразу несколько регистров.

Другой пример применения паттерна компоновщик — финансовые программы, когда инвестиционный портфель состоит из нескольких отдельных активов. Можно поддержать сложные агрегаты активов, если реализовать портфель в виде компоновщика, согласованного с интерфейсом каждого актива [BE93].

Паттерн команда (275) описывает возможности компоновки и упорядочения объектов `Command` с помощью класса компоновщика `MacroCommand`.

Родственные паттерны

Отношение «компонент — родитель» используется в паттерне цепочка обязанностей (263).

Паттерн декоратор часто применяется совместно с компоновщиком. Когда декораторы и компоновщики используются вместе, у них обычно бывает общий родительский класс. Поэтому декораторам придется поддержать интерфейс компонентов такими операциями, как `Add`, `Remove` и `GetChild`.

Паттерн приспособленец (231) позволяет совместно использовать компоненты, но ссылаться на своих родителей они уже не могут.

Итератор (302) можно использовать для обхода составных объектов.

Посетитель (379) локализует операции и поведение, которые в противном случае пришлось бы распределять между классами `Composite` и `Leaf`.

ПАТТЕРН DECORATOR (ДЕКОРАТОР)

Название и классификация паттерна

Декоратор — паттерн, структурирующий объекты.

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Другие названия

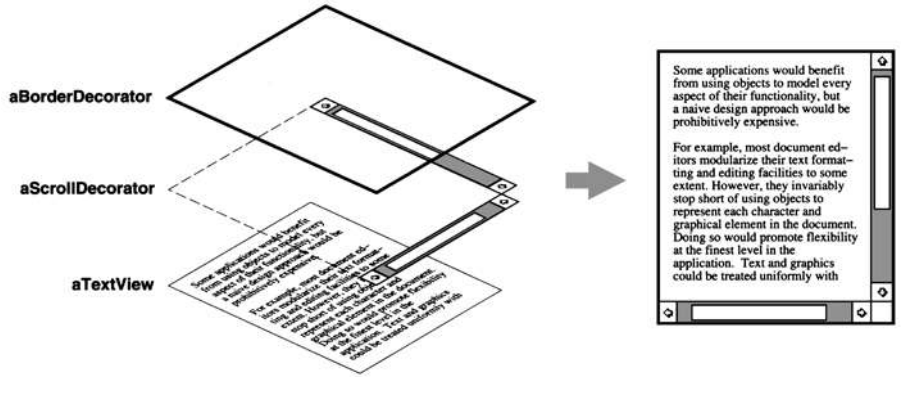
`Wrapper` (обертка).

Мотивация

Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, библиотека для построения графических интерфейсов пользователя должна «уметь» добавлять новое свойство, скажем, рамку или новое поведение (например, возможность прокрутки к любому элементу интерфейса).

Новые обязанности можно добавить с помощью наследования. При наследовании классу с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако такое решение недостаточно гибкое из-за того, что рамка будет выбираться статически. Клиент не может управлять тем, когда и как компоненты будут декорироваться обрамлением.

Другое, более гибкое решение — поместить компонент в другой объект, называемый *декоратором*, который как раз и добавляет рамку. Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента. Декоратор переадресует запросы внутреннему компоненту, но может выполнять и дополнительные действия (например, рисовать рамку) до или после переадресации. Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым неограниченное число новых обязанностей.

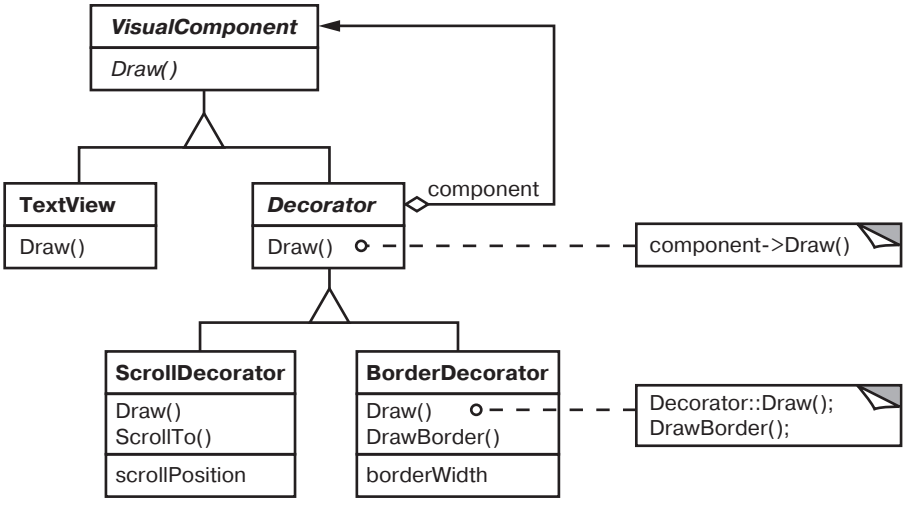


Предположим, что имеется объект класса `TextView`, который отображает текст в окне. По умолчанию `TextView` не имеет полос прокрутки, поскольку они не всегда нужны. Но при необходимости их удастся добавить с помощью декоратора `ScrollDecorator`. Допустим, что еще мы хотим добавить жирную сплошную рамку вокруг объекта `TextView`. Здесь может помочь декоратор `BorderDecorator`. Мы просто компонуем оба декоратора с `BorderDecorator` для получения искомого результата.

Ниже на схеме показано, как композиция объекта `TextView` с объектами `BorderDecorator` и `ScrollDecorator` формирует элемент для ввода текста, окруженный рамкой и снабженный полосой прокрутки:

Классы `ScrollDecorator` и `BorderDecorator` являются подклассами `Decorator` — абстрактного класса, который представляет визуальные компоненты, применяемые для оформления других визуальных компонентов.

`VisualComponent` — это абстрактный класс для представления визуальных объектов. В нем определен интерфейс для рисования и обработки событий. Отметим, что класс `Decorator` просто переадресует запросы на рисование своему компоненту, а его подклассы могут расширять эту операцию.



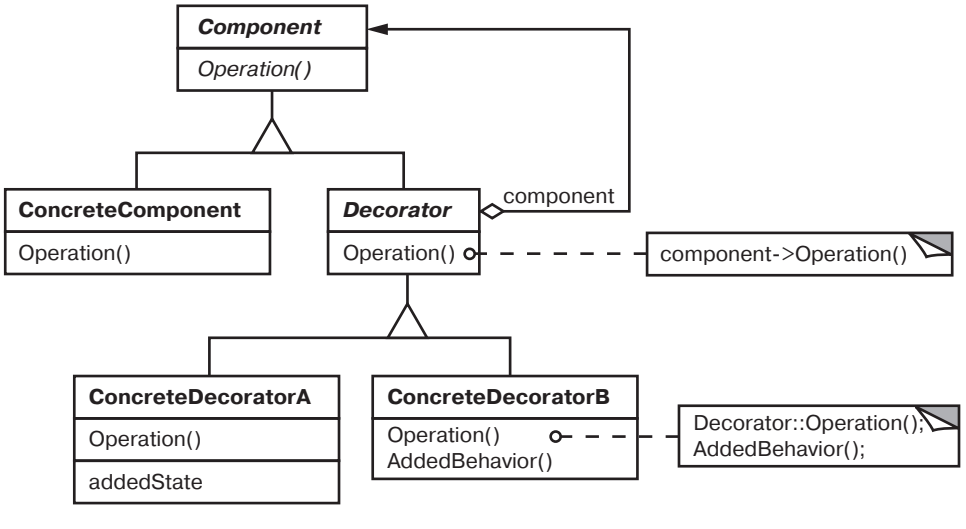
Подклассы `Decorator` могут добавлять любые операции для обеспечения необходимой функциональности. Так, операция `ScrollTo` объекта `ScrollDecorator` позволяет другим объектам выполнять прокрутку, если им известно о присутствии объекта `ScrollDecorator` в интерфейсе. Важная особенность этого паттерна состоит в том, что декораторы могут употребляться везде, где возможно появление самого объекта `VisualComponent`. При этом клиент не может отличить декорированный объект от недекорированного, а значит, и никоим образом не зависит от наличия или отсутствия декоративных элементов.

Применимость

Основные условия для применения паттерна декоратор:

- динамическое, прозрачное для клиентов добавление обязанностей объектам (не затрагивающее другие объекты);
- реализация обязанностей, которые могут быть сняты с объекта;
- расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к стремительному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Структура



Участники

- **Component** (`VisualComponent`) — компонент:
  - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- **ConcreteComponent** (`TextView`) — конкретный компонент:
  - определяет объект, на который возлагаются дополнительные обязанности;
- **Decorator** — декоратор:
  - хранит ссылку на объект `Component` и определяет интерфейс, соответствующий интерфейсу `Component`;
- **ConcreteDecorator** (`BorderDecorator`, `ScrollDecorator`) — конкретный декоратор:
  - возлагает дополнительные обязанности на компонент.

Отношения

`Decorator` переадресует запросы объекту `Component`. Может выполнять и дополнительные операции до и после переадресации.

Результаты

У паттерна декоратор есть, по крайней мере, два плюса и два минуса:

- *большая гибкость, нежели у статического наследования.* Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. С другой стороны, при использовании наследования требуется создавать новый класс для каждой дополнительной обязанности (например, `BorderedScrollableTextView`, `BorderedTextView`), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет формировать произвольные комбинации обязанностей.

  Декораторы также позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект `TextView` двойной рамкой, нужно просто добавить два декоратора `BorderDecorators`. Двойное наследование классу `Border` в лучшем случае чревато ошибками;

- *позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.* Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, вы можете определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не перегружается неиспользуемыми функциями. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции;
- *декоратор и его компонент не идентичны.* Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду;
- *множество мелких объектов.* При использовании в проекте паттерна декоратор нередко формируется система, составленная из большого числа мелких объектов, похожих друг на друга. Такие объекты различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя такие системы легко настраиваются проектировщиком, хорошо разбирающимся в их строении, изучать и отлаживать их очень тяжело.

Реализация

При применении паттерна декоратор следует учитывать ряд аспектов:

- *соответствие интерфейсов.* Интерфейс декоратора должен соответствовать интерфейсу декорируемого компонента. Поэтому классы `ConcreteDecorator` должны наследовать общему классу (по крайней мере, в C++);
- *отсутствие абстрактного класса Decorator.* Нет необходимости определять абстрактный класс `Decorator`, если вы собираетесь добавить всего одну обязанность. Так часто происходит, когда вы работаете с уже существующей иерархией классов, а не проектируете новую. В таком случае ответственность за переадресацию запросов, которую обычно несет класс `Decorator`, можно возложить непосредственно на `ConcreteDecorator`;
- *облегченные классы Component.* Чтобы можно было гарантировать соответствие интерфейсов, компоненты и декораторы должны наследовать общему классу `Component`. Важно, чтобы этот класс был настолько легким, насколько возможно. Иными словами, он должен определять

интерфейс, а не хранить данные. Определение представления данных должно быть передано в подклассы; в противном случае декораторы могут стать весьма тяжеловесными, и применять их в большом количестве будет накладно. Включение большого числа функций в класс `Component` также увеличивает вероятность, что конкретным подклассам придется платить за то, что им не нужно;

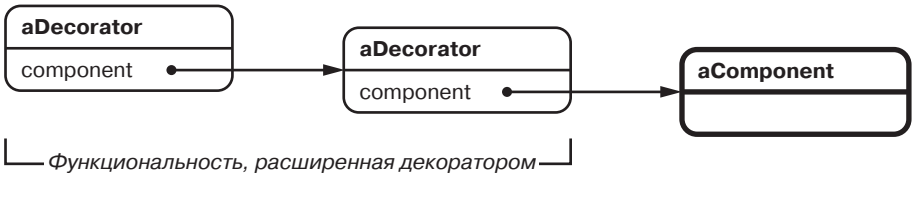
- *изменение облика, а не внутреннего устройства объекта*. Декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива — изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн стратегия (362).

  Стратегии лучше подходят в ситуациях, когда класс `Component` уже достаточно тяжел, так что применение паттерна декоратор обходится слишком дорого. В паттерне стратегия компоненты передают часть своей функциональности отдельному объекту-стратегии, поэтому изменить или расширить поведение компонента допустимо, заменив этот объект.

  Например, мы можем поддержать разные стили рамок, поручив рисование рамки специальному объекту `Border`. Объект `Border` является примером объекта-стратегии: в данном случае он инкапсулирует стратегию рисования рамки. Число стратегий может быть любым, поэтому эффект такой же, как от рекурсивной вложенности декораторов.

  Например, в системах MacApp 3.0 [App89] и Bedrock [Sym93a] графические компоненты, называемые *представлениями* (views), хранят список объектов-*оформителей* (adorner), которые могут добавлять различные оформления вроде границ к виду. Если к представлению присоединены такие объекты, он дает им возможность выполнить свои функции. MacApp и Bedrock вынуждены предоставить доступ к этим операциям, поскольку класс `View` весьма тяжеловесен. Было бы слишком расточительно использовать полномасштабный объект этого класса только для того, чтобы добавить рамку.

  Поскольку паттерн декоратор изменяет лишь внешний облик компонента, последнему ничего не надо «знать» о своих декораторах, то есть декораторы прозрачны для компонента.



В случае стратегий самому компоненту известно о возможных расширениях. Поэтому он должен располагать информацией обо всех стратегиях и ссылаться на них.

При использовании подхода, основанного на стратегиях, может возникнуть необходимость в модификации компонента, чтобы он соответствовал новому расширению. С другой стороны, у стратегии может быть свой собственный специализированный интерфейс, тогда как интерфейс декоратора должен повторять интерфейс компонента. Например, стратегии рисования рамки необходимо определить всего лишь интерфейс для этой операции (`DrawBorder`, `GetWidth` и т. д.), то есть класс стратегии может быть легким, несмотря на тяжеловесность компонента.

Системы MacApp и Bedrock применяют такой подход не только для оформления представлений, но и для расширения особенностей поведения объектов, связанных с обработкой событий. В обеих системах представление ведет список объектов поведения, которые могут модифицировать и перехватывать события. Каждому зарегистрированному объекту поведения представление предоставляет возможность обработать событие до того, как оно будет передано незарегистрированным объектам такого рода, за счет чего достигается переопределение поведения. Можно, например, декорировать вид специальной поддержкой работы с клавиатурой, если зарегистрировать объект поведения, который перехватывает и обрабатывает события нажатия клавиш.

Пример кода

Следующий пример показывает, как декораторы пользовательского интерфейса реализуются в программе на C++. Предполагается, что класс компонента называется `VisualComponent`:

```
class VisualComponent {
public:
    VisualComponent()
    virtual void Draw();
```

```
    virtual void Resize();
    // ...
};
```

Определим подкласс класса `VisualComponent` с именем `Decorator`, от которого затем породим подклассы, реализующие различные оформления:

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

Объект класса `Decorator` декорирует объект `VisualComponent`, на который ссылается переменная экземпляра `_component`, инициализируемая в конструкторе. Для каждой операции в интерфейсе `VisualComponent` в классе `Decorator` определена реализация по умолчанию, передающая запросы объекту, на который ведет ссылка `_component`:

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Подклассы `Decorator` определяют специализированные операции. Например, класс `BorderDecorator` добавляет к своему внутреннему компоненту рамку. `BorderDecorator` — это подкласс `Decorator`, где операция `Draw` замещена так, что рисует рамку. В этом классе определена также закрытая вспомогательная операция `DrawBorder`, которая, собственно, и изображает рамку. Реализации всех остальных операций этот подкласс наследует от `Decorator`:

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);
    virtual void Draw();
private:
    void DrawBorder(int);
```

```
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

Подклассы `ScrollDecorator` и `DropShadowDecorator`, которые добавят визуальному компоненту возможность прокрутки и оттенения, реализуются аналогично.

Теперь экземпляры этих классов можно скомпоновать для получения различных оформлений. Ниже показано, как использовать декораторы для создания прокручиваемого компонента `TextView` с рамкой.

Во-первых, нужно каким-то образом поместить визуальный компонент в оконный объект. Предположим, что в классе `Window` для этой цели имеется операция `SetContents`:

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

Теперь можно создать поле для ввода текста и окно, в котором будет находиться это поле:

```
Window* window = new Window;
TextView* textView = new TextView;
```

`TextView` является подклассом `VisualComponent`, значит, мы могли бы поместить его в окно:

```
window->SetContents(textView);
```

Но нам нужно поле ввода с рамкой и возможностью прокрутки, поэтому перед размещением в окне его необходимо соответствующим образом оформить:

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Поскольку класс `Window` обращается к своему содержимому только через интерфейс `VisualComponent`, то ему неизвестно о присутствии декоратора. Клиент при желании может сохранить ссылку на само поле ввода, если ему нужно работать с ним непосредственно — например, вызывать операции, не входящие в интерфейс `VisualComponent`. Клиенты, которым важна идентичность объекта, также должны обращаться к нему напрямую.

Известные применения

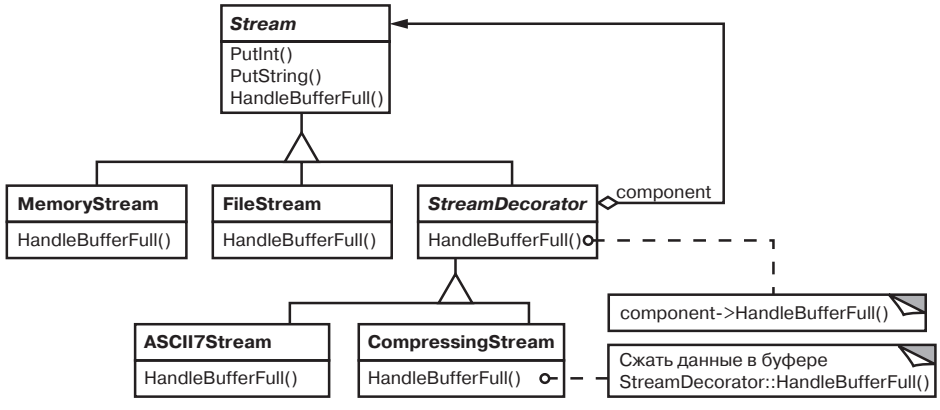
Во многих библиотеках для построения объектно-ориентированных интерфейсов пользователя декораторы применяются для добавления к виджетам графических оформлений. В качестве примеров можно назвать InterViews [LVC89, LCI+92], ET++ [WGM88] и библиотеку классов ObjectWorks\Smalltalk [Par90]. Более экзотические варианты применения паттерна `декоратор` — это класс `DebuggingGlyph` из библиотеки InterViews и `PassivityWrapper` из ParcPlace Smalltalk. `DebuggingGlyph` выводит отладочную информацию до и после того, как переадресует запрос на размещение своему компоненту. Эта информация может быть полезной для анализа и отладки стратегии размещения объектов в сложной композиции. Класс `PassivityWrapper` позволяет разрешить или запретить взаимодействие компонента с пользователем.

Но применение паттерна `декоратор` никоим образом не ограничивается графическими интерфейсами пользователя, как показывает следующий пример, основанный на потоковых классах из каркаса ET++ [WGM88].

Поток является фундаментальной абстракцией для большинства средств ввода/вывода. Он может предоставлять интерфейс для преобразования объектов в последовательность байтов или символов. Это позволяет записать объект в файл или буфер в памяти и впоследствии извлечь его оттуда. Самый очевидный способ сделать это — определить абстрактный класс `Stream` с подклассами `MemoryStream` и `FileStream`. Предположим, однако, что вам также хотелось бы иметь возможность:

- сжимать данные в потоке с применением различных алгоритмов (кодирование с переменной длиной строки, алгоритм Лемпеля — Зива и т. д.);
- преобразовывать данные в 7-битные символы кода ASCII для передачи по каналу связи.

Паттерн `декоратор` позволяет весьма элегантно добавить такие обязанности потокам. На схеме ниже показано одно из возможных решений задачи.



Абстрактный класс `Stream` имеет внутренний буфер и предоставляет операции для помещения данных в поток (PutInt, PutString). Как только буфер заполняется, `Stream` вызывает абстрактную операцию `HandleBufferFull`, которая выполняет реальное перемещение данных. В классе `FileStream` эта операция замещается так, что буфер записывается в файл.

Ключевую роль здесь играет класс `StreamDecorator`. Именно в нем хранится ссылка на тот поток-компонент, которому переадресуются все запросы. Подклассы `StreamDecorator` замещают операцию `HandleBufferFull` и выполняют дополнительные действия, перед тем как вызвать реализацию этой операции в классе `StreamDecorator`.

Например, подкласс `CompressingStream` сжимает данные, а `ASCII7Stream` преобразует их в 7-битный код ASCII. Теперь, для того чтобы создать объект `FileStream`, который *одновременно* сжимает данные и преобразует результат в 7-битный код, достаточно просто декорировать `FileStream` с использованием `CompressingStream` и `ASCII7Stream`:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
         new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Родственные паттерны

Адаптер (171): декоратор изменяет только обязанности объекта, не меняя интерфейса, а адаптер придает объекту совершенно новый интерфейс.

Компоновщик (196): декоратор можно считать вырожденным случаем составного объекта, у которого есть только один компонент. Однако декоратор добавляет новые обязанности, агрегирование объектов не является его целью.

Стратегия (362): декоратор позволяет изменить внешний облик объекта, стратегия — его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

ПАТТЕРН FACADE (ФАСАД)

Название и классификация паттерна

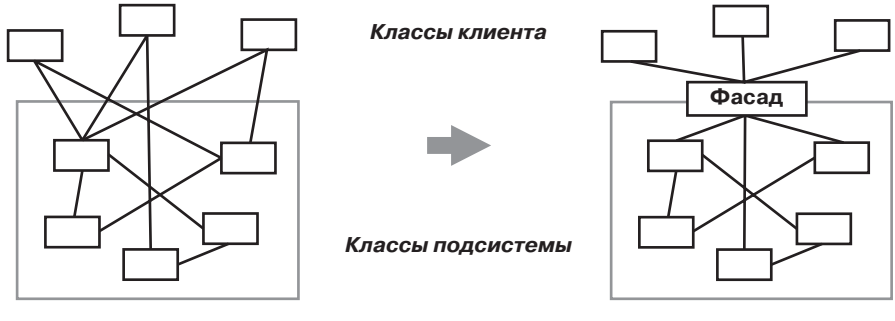
Фасад — паттерн, структурирующий объекты.

Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Мотивация

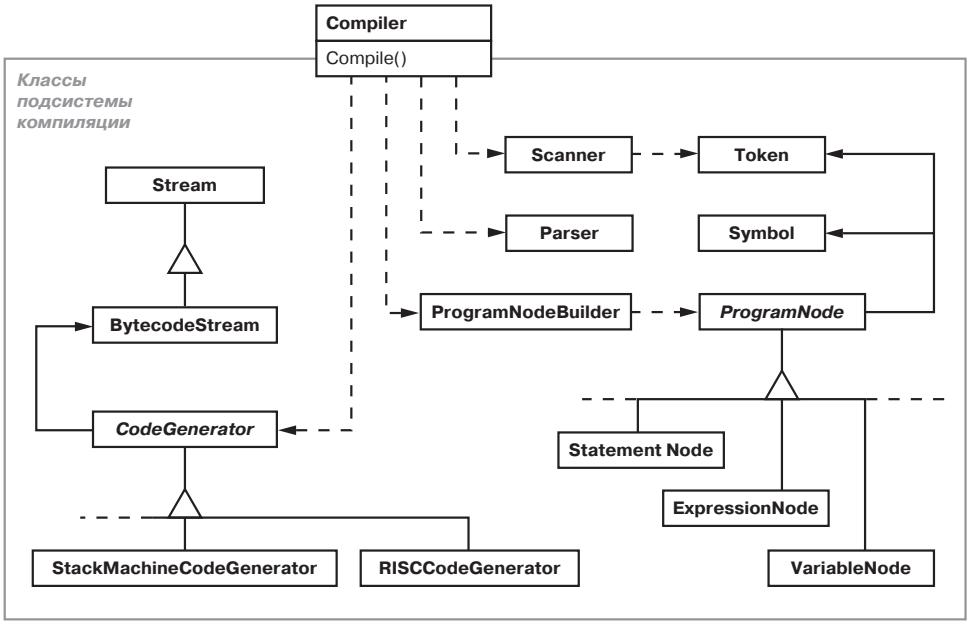
Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования — свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи — введение объекта фасад, который предоставляет единый упрощенный интерфейс к более сложным системным средствам.



Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В эту подсистему входят классы, реализующие компилятор — такие как `Scanner` (лексический

анализатор), `Parser` (синтаксический анализатор), `ProgramNode` (узел программы), `BytecodeStream` (поток байтовых кодов) и `ProgramNodeBuilder` (строитель узла программы). Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерирование кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса в подсистеме компиляции только усложняет задачу.

Чтобы предоставить интерфейс более высокого уровня, изолирующий клиент от этих классов, в подсистему компиляции включен также класс `Compiler` (компилятор). Он определяет унифицированный интерфейс ко всей функциональности компилятора. Класс `Compiler` выступает в роли фасада: он предоставляет простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Фасад компилятора упрощает работу большинства программистов, не скрывая низкоуровневую функциональность для тех немногих, кому она нужна.

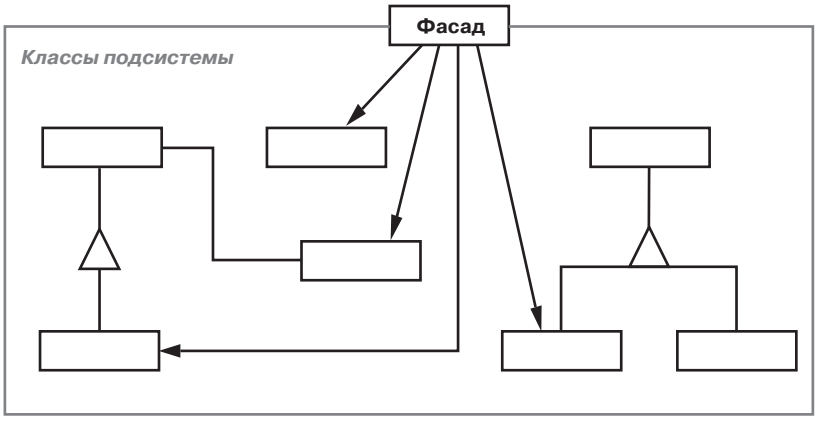


Применимость

Основные условия для применения паттерна фасад:

- *предоставление простого интерфейса к сложной подсистеме*. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к тому, что в системе появляется множество мелких классов. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему тем клиентам, которым не нужно ее настраивать, становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те клиенты, которым нужны более широкие возможности настройки, могут напрямую обратиться к тому, что находится за фасадом;
- *многочисленные зависимости между клиентами и классами реализации абстракции*. Фасад позволяет отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует независимости подсистем и повышению уровня переносимости;
- *требуется разложить подсистему на отдельные уровни*. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимости можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Структура



Участники

- **Facade** (`Compiler`) — фасад:
  - «знает», каким классам подсистемы адресовать запрос;
  - делегирует запросы клиентов подходящим объектам внутри подсистемы;
- **Классы подсистемы** (`Scanner`, `Parser`, `ProgramNode` и т. д.):
  - реализуют функциональность подсистемы;
  - выполняют работу, порученную объектом Facade;
  - ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

Отношения

Клиенты общаются с подсистемой, посылая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы.

Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам подсистемы.

Результаты

Основные достоинства паттерна фасад:

- он изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;
- позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на уровни и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо.

  Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Хочется, конечно, чтобы время, уходящее на перекомпиляцию после изменения классов подсистемы, было минимальным. Сокращение числа зависимостей за счет фасадов может

уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные;

- фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Реализация

При реализации фасада следует обратить внимание на следующие вопросы:

- *ослабление связанности клиента с подсистемой.* Степень связанности можно значительно уменьшить, если сделать класс `Facade` абстрактным. Его конкретные подклассы будут соответствовать различным реализациям подсистемы. Тогда клиенты смогут взаимодействовать с подсистемой через интерфейс абстрактного класса `Facade`. Абстрактное связывание изолирует клиентов от информации о том, какая реализация подсистемы используется.

  Вместо порождения подклассов можно сконфигурировать объект `Facade` различными объектами подсистем. Для настройки фасада достаточно заменить один или несколько таких объектов;

- *открытые и закрытые классы подсистем.* Подсистема похожа на класс в том отношении, что у обоих есть интерфейсы и оба что-то инкапсулируют. Класс инкапсулирует состояние и операции, а подсистема — классы. И если полезно различать открытый и закрытый интерфейсы класса, то не менее разумно говорить об открытом и закрытом интерфейсах подсистемы.

  Открытый интерфейс подсистемы состоит из классов, к которым имеют доступ все клиенты; закрытый интерфейс доступен только для расширения подсистемы. Класс `Facade`, конечно же, является частью открытого интерфейса, но это не единственная часть. Другие классы подсистемы также могут быть открытыми. Например, в подсистеме компиляции классы `Parser` и `Scanner` являются частью открытого интерфейса.

  Объявлять классы подсистемы закрытыми бывает полезно, но такая возможность поддерживается немногими объектно-ориентированными языками. И в C++, и в Smalltalk для классов традиционно использовалось глобальное пространство имен. Однако недавно комитет по стандарти-

зации C++ добавил к языку пространства имен [Str94], и это позволило предоставить доступ только к открытым классам подсистемы.

Пример кода

Рассмотрим более подробно, как создать фасад для подсистемы компиляции.

В подсистеме компиляции определен класс `BytecodeStream`, который реализует поток объектов `Bytecode`. Объект `Bytecode` инкапсулирует байтовый код, с помощью которого описываются машинные команды. В этой же подсистеме определен еще класс `Token` для объектов, инкапсулирующих лексемы языка программирования.

Класс `Scanner` принимает на входе поток символов и генерирует поток лексем, по одной каждый раз:

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

Класс `Parser` использует класс `ProgramNodeBuilder` для построения дерева разбора из лексем, возвращенных классом `Scanner`:

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

`Parser` вызывает `ProgramNodeBuilder` для инкрементного построения дерева. Взаимодействие этих классов описывается паттерном строитель (124):

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

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;
```

```
    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...

    ProgramNode* GetRootNode();
private:
    ProgramNode* _node;
};
```

Дерево разбора состоит из экземпляров подклассов класса `ProgramNode`, таких как `StatementNode`, `ExpressionNode` и т. д. Иерархия классов `ProgramNode` — это пример паттерна компоновщик (196). Класс `ProgramNode` определяет интерфейс для выполнения операций с узлом программы и его потомками, если таковые имеются:

```
class ProgramNode {
public:
    // операции с узлом программы
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // операции с потомками
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    // ...

    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

Операция `Traverse` (обход) принимает объект `CodeGenerator` (кодогенератор) в качестве параметра. Подклассы `ProgramNode` используют этот объект для генерирования машинного кода в форме объектов `Bytecode`, которые помещаются в поток `BytecodeStream`. Класс `CodeGenerator` описывается паттерном посетитель (379):

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    // ...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& _output;
};
```

У `CodeGenerator` есть подклассы, генерирующие машинный код для различных аппаратных архитектур — например `StackMachineCodeGenerator` и `RISCCodeGenerator`.

Каждый подкласс `ProgramNode` реализует операцию `Traverse` и обращается к ней для обхода своих потомков. В свою очередь, каждый потомок рекурсивно делает то же самое для своих потомков. Например, в подклассе `ExpressionNode` (узел выражения) операция `Traverse` определена так:

```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

Классы, о которых мы говорили до сих пор, составляют подсистему компиляции. А теперь введем класс `Compiler`, который будет служить фасадом, позволяющим собрать все эти фрагменты воедино. Класс `Compiler` предоставляет простой интерфейс для компилирования исходного текста и генерирования кода для конкретной машины:

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

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
```

```
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

В этой реализации жестко «зашит» тип кодогенератора, поэтому программисту не нужно явно задавать целевую архитектуру. Это может быть вполне разумно, когда такая архитектура всего одна. Если же это не так, можно было бы изменить конструктор класса `Compiler`, чтобы он принимал объект `CodeGenerator` в качестве параметра. Тогда программист указывал бы, каким генератором пользоваться при создании экземпляра `Compiler`. Фасад компилятора можно параметризовать и другими участниками, скажем, объектами `Scanner` и `ProgramNodeBuilder`, что повышает гибкость, но в то же время сводит на нет основную цель фасада — предоставление упрощенного интерфейса для наиболее распространенного случая.

Известные применения

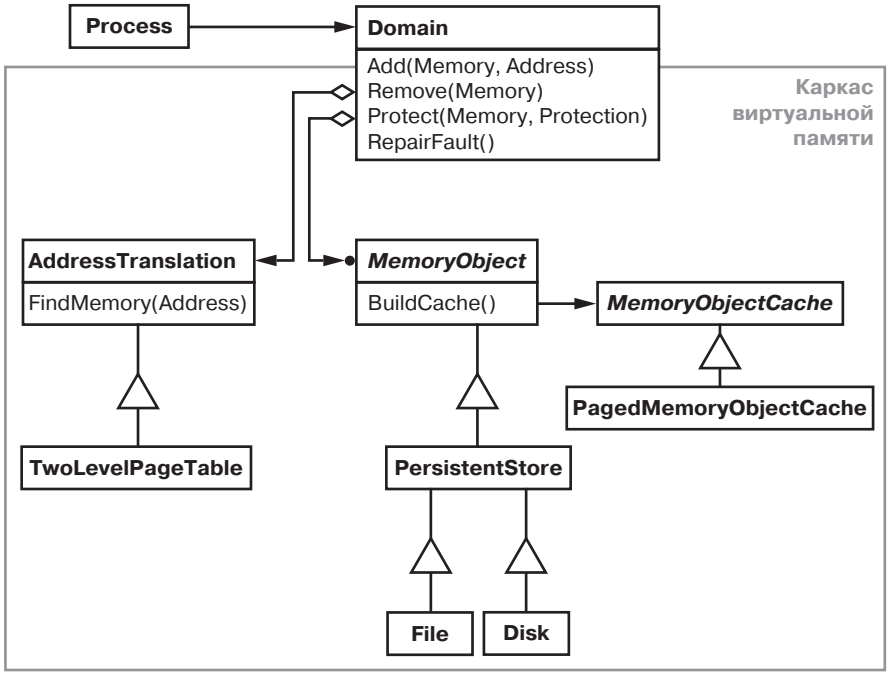
Пример с компилятором в разделе «Пример кода» навеян идеями из системы компиляции языка ObjectWorks\Smalltalk [Par90].

В каркасе ET++ [WGM88] приложение может иметь встроенные средства инспектирования объектов во время выполнения. Они реализуются в отдельной подсистеме, включающей класс фасада с именем `ProgrammingEnvironment`. Этот фасад определяет такие операции, как `InspectObject` и `InspectClass` для доступа к инспекторам.

Приложение, написанное в среде ET++, может также запретить поддержку инспектирования. В таком случае класс `ProgrammingEnvironment` реализует соответствующие запросы как пустые операции, не делающие ничего. Только подкласс `ETProgrammingEnvironment` реализует эти операции так, что они отображают окна соответствующих инспекторов. Приложению неизвестно, доступно инспектирование или нет. Здесь мы встречаем пример абстрактной связанности между приложением и подсистемой инспектирования.

В операционной системе Choices [CIRM93] фасады используются для составления одного каркаса из нескольких. Ключевыми абстракциями в системе Choices являются процессы, память и адресные пространства. Для каждой из них есть соответствующая подсистема, реализованная в виде каркаса. Это обеспечивает поддержку переноса Choices на разные аппаратные плат-

формы. У двух таких подсистем есть «представители», то есть фасады. Они называются `FileSystemInterface` (подсистема хранения данных) и `Domain` (адресные пространства).



Например, для каркаса виртуальной памяти фасадом служит класс `Domain`, представляющий адресное пространство. Он обеспечивает отображение между виртуальными адресами и смещениями объектов в памяти, файле или на устройстве длительного хранения. Базовые операции класса `Domain` поддерживают добавление объекта в память по указанному адресу, удаление объекта из памяти и обработку ошибок отсутствия страниц.

Как видно из вышеприведенной диаграммы, внутри подсистемы виртуальной памяти используются следующие компоненты:

- `MemoryObject` представляет объекты данных;
- `MemoryObjectCache` кэширует данные из объектов `MemoryObjects` в физической памяти. `MemoryObjectCache` — это не что иное, как объект стратегия (362), в котором локализована политика кэширования;
- `AddressTranslation` инкапсулирует особенности оборудования трансляции адресов.

Операция `RepairFault` вызывается при возникновении ошибки из-за отсутствия страницы в памяти. `Domain` находит объект в памяти по адресу, по которому произошла ошибка, и делегирует операцию `RepairFault` кэшу, ассоциированному с этим объектом. Поведение объектов `Domain` можно настроить, заменив их компоненты.

Родственные паттерны

Паттерн абстрактная фабрика (113) допустимо использовать вместе с фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, независимым от этих подсистем. Абстрактная фабрика может выступать и как альтернатива фасаду, чтобы скрыть платформеннозависимые классы.

Паттерн посредник (319) аналогичен фасаду в том смысле, что абстрагирует функциональность существующих классов. Однако назначение посредника — абстрагировать произвольное взаимодействие между «сотрудничающими» объектами. Часто он централизует функциональность, не присущую ни одному из них. Коллеги посредника знают о его существовании и обмениваются информацией именно с ним, а не напрямую между собой. С другой стороны, фасад просто абстрагирует интерфейс объектов подсистемы, чтобы ими было проще пользоваться. Он не определяет новой функциональности, и классам подсистемы ничего неизвестно о его существовании.

Обычно в системе должен существовать только один фасад, поэтому объекты фасадов часто бывают одиночками (157).

ПАТТЕРН FLYWEIGHT (ПРИСПОСОБЛЕНЕЦ)

Название и классификация паттерна

Приспособленец — паттерн, структурирующий объекты.

Назначение

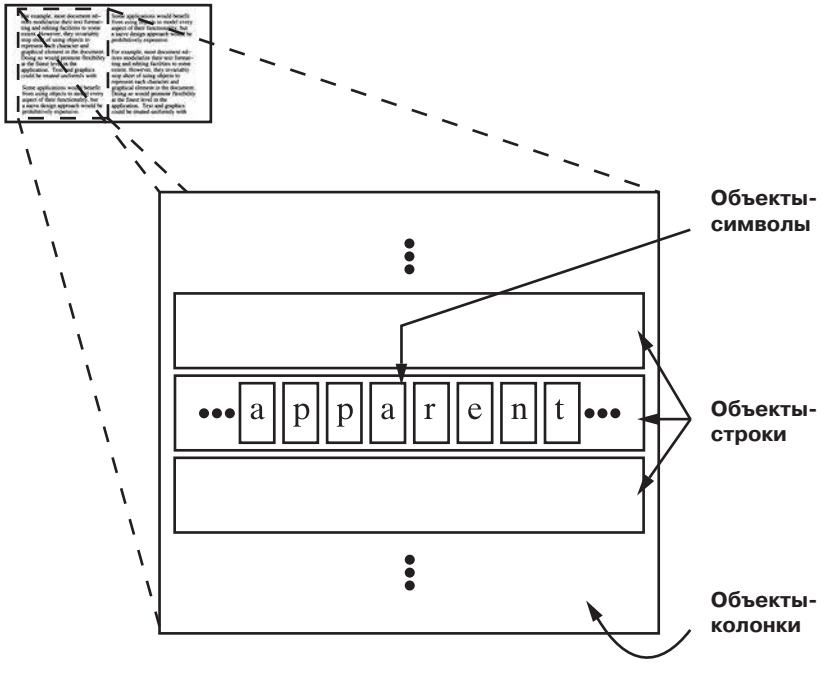
Применяет совместное использование для эффективной поддержки множества мелких объектов.

Мотивация

В некоторых приложениях использование объектов могло бы быть очень полезным, но прямолинейная реализация оказывается недопустимо расточительной.

Например, в большинстве редакторов документов имеются средства форматирования и редактирования текстов, в той или иной степени модульные. Объектно-ориентированные редакторы обычно применяют объекты для представления таких встроенных элементов, как таблицы и рисунки. Но они не применяют объекты для представления каждого символа, несмотря на то что это увеличило бы гибкость на самых нижних уровнях приложения — ведь тогда символы и встроенные элементы можно было бы прорисовывать и форматировать по единым принципам, и для поддержки новых наборов символов не пришлось бы как-либо затрагивать остальные функции редактора. Вдобавок общая структура приложения отражала бы физическую структуру документа. На следующей диаграмме показано, как редактор документов мог бы воспользоваться объектами для представления символов.

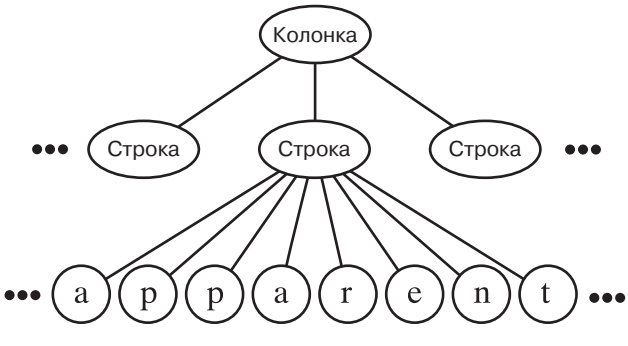
У такого дизайна есть один недостаток — затраты ресурсов. Даже в документе скромных размеров было бы несколько сотен тысяч объектов-символов, а это привело бы к расходованию огромного объема памяти и неприемлемым затратам во время выполнения. Паттерн приспособленец показывает, как совместно использовать очень мелкие объекты без недопустимо высоких затрат.



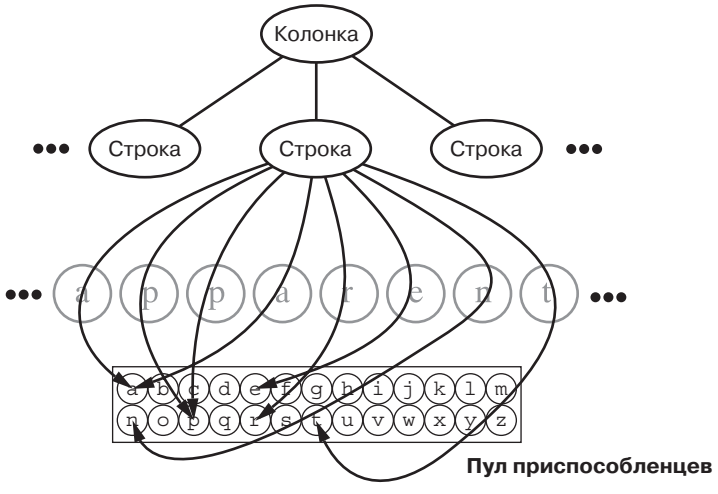
Приспособленец — это совместно используемый объект, который можно задействовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не используется совместно. Приспособленцы не могут делать предположений о контексте, в котором работают. Ключевая идея здесь — различие между *внутренним* и *внешним* состояниями. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может использоваться совместно. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому совместно не используется. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами. Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа — это внутреннее состояние, а все остальное — внешнее.

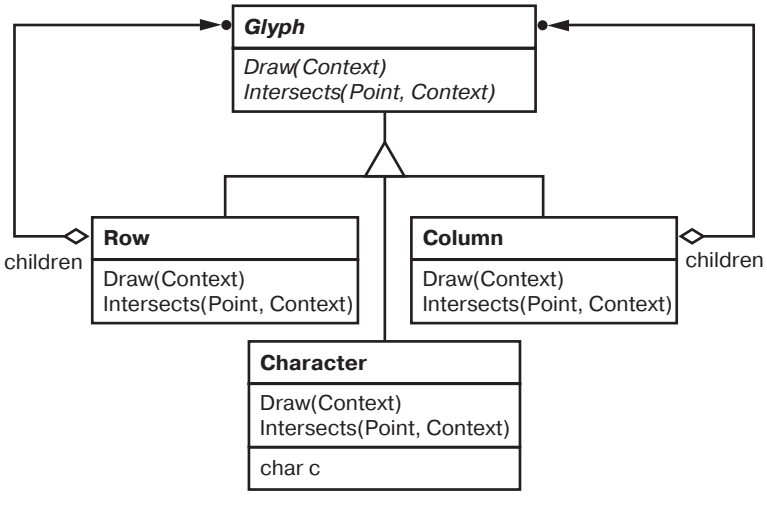
На логическом уровне для каждого вхождения данного символа в документ существует объект.



Однако на физическом уровне существует лишь по одному объекту-приспособленцу для каждого символа, который появляется в различных контекстах в структуре документа. Каждое вхождение данного объекта-символа ссылается на один и тот же экземпляр в совместно используемом пуле объектов-приспособленцев.



Ниже изображена структура класса для этих объектов. `Glyph` — это абстрактный класс для представления графических объектов (некоторые из них могут быть приспособленцами). Операции, которые могут зависеть от внешнего состояния, передают его в качестве параметра. Например, операциям `Draw` (рисование) и `Intersects` (пересечение) должно быть известно, в каком контексте встречается глиф, иначе они не смогут выполнить то, что от них требуется.



Приспособленец, представляющий букву «а», содержит только соответствующий ей код; ни положение, ни шрифт буквы ему хранить не надо. Клиенты передают приспособленцу всю зависящую от контекста информацию, которая нужна, чтобы он мог изобразить себя. Например, глифу `Row` известно, где его потомки должны себя вывести, чтобы быть выстроенными в ряд по горизонтали. Поэтому вместе с запросом на рисование он может передавать каждому потомку координаты.

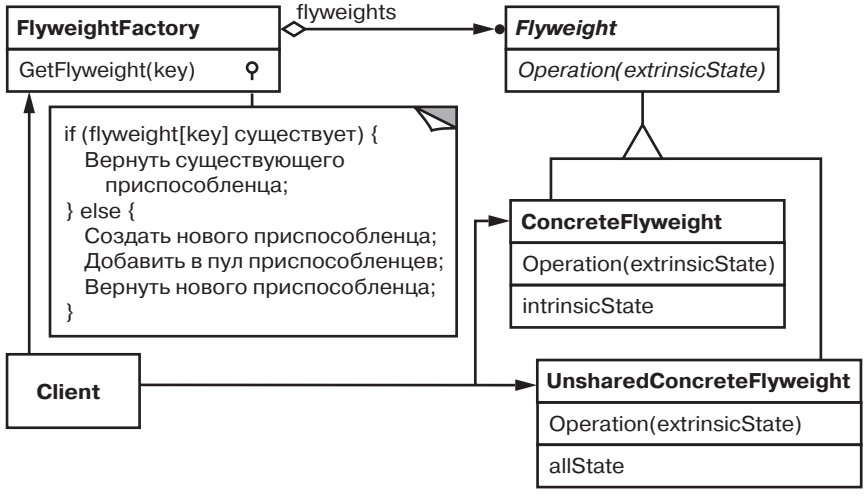
Поскольку число различных объектов-символов гораздо меньше числа символов в документе, то и общее количество объектов существенно меньше, чем было бы при простой реализации. Документ, в котором все символы изображаются одним шрифтом и цветом, создаст порядка 100 объектов-символов (это примерно равно числу кодов в таблице ASCII) независимо от своего размера. А поскольку в большинстве документов применяется не более десятка различных комбинаций шрифта и цвета, то на практике эта величина возрастет несущественно, поэтому абстракция объекта становится применимой и к отдельным символам.

Применимость

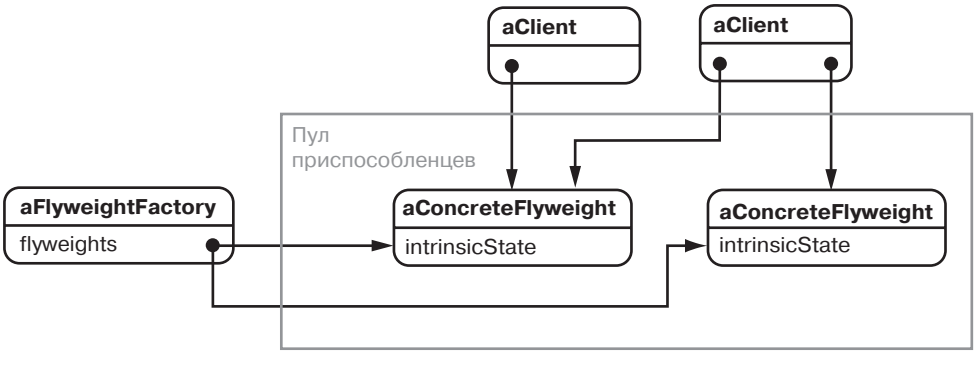
Эффективность паттерна приспособленец во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены *все* нижеперечисленные условия:

- в приложении используется большое число объектов;
- из-за этого затраты на хранение высоки;
- большую часть состояния объектов можно вынести вовне;
- многие группы объектов можно заменить относительно небольшим количеством совместно используемых объектов, поскольку внешнее состояние вынесено;
- приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут использоваться совместно, то проверка на идентичность возвратит признак истинности для концептуально различных объектов.

Структура



На следующей схеме показано, как организуется совместное использование приспособленцев.



Участники

- **Flyweight** (`Glyph`) — приспособленец:
  - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- **ConcreteFlyweight** (`Character`) — конкретный приспособленец:
  - реализует интерфейс класса `Flyweight` и добавляет при необходимости внутреннее состояние. Объект класса `ConcreteFlyweight` должен быть

совместно используемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;

- **UnsharedConcreteFlyweight** (`Row`, `Column`) — конкретный приспособленец, не используемый совместно:
  - не все подклассы `Flyweight` обязательно должны быть совместно используемыми. Интерфейс `Flyweight` допускает совместное использование, но не навязывает его. Часто у объектов `UnsharedConcreteFlyweight` на некотором уровне структуры приспособленца есть потомки в виде объектов класса `ConcreteFlyweight`, как, например, у объектов классов `Row` и `Column`;
- **FlyweightFactory** — фабрика приспособленцев:
  - создает объекты-приспособленцы и управляет ими;
  - обеспечивает совместное использование приспособленцев. Когда клиент запрашивает приспособленца, объект `FlyweightFactory` предоставляет существующий экземпляр или создает новый, если готового еще нет;
- **Client** — клиент:
  - хранит ссылки на одного или нескольких приспособленцев;
  - вычисляет или хранит внешнее состояние приспособленцев.

Отношения

- Состояние, необходимое приспособленцу для нормальной работы, классифицируется на внутреннее или внешнее. Внутреннее состояние хранится в самом объекте `ConcreteFlyweight`. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций;
- клиенты не должны создавать экземпляры класса `ConcreteFlyweight` напрямую, а могут получать их только от объекта `FlyweightFactory`. Это позволит гарантировать корректное совместное использование.

Результаты

При использовании приспособленцев возможны затраты на передачу, поиск или вычисление внутреннего состояния на стадии выполнения, особенно если раньше оно хранилось как внутреннее. Однако такие затраты с лихвой компенсируются экономией памяти за счет совместного использования объектов-приспособленцев.

Экономия памяти обусловлена несколькими причинами:

- уменьшение общего числа экземпляров;
- сокращение объема памяти, необходимого для хранения внутреннего состояния;
- вычисление, а не хранение внешнего состояния (если это действительно так).

Чем выше степень совместного использования приспособленцев, тем существеннее экономия. С увеличением объема совместного состояния экономия также возрастает. Самого большого эффекта удается добиться, когда суммарный объем внутренней и внешней информации о состоянии велик, а внешнее состояние вычисляется, а не хранится. Тогда совместное использование уменьшает стоимость хранения внутреннего состояния, а за счет вычислений сокращается память, отводимая под внешнее состояние.

Паттерн приспособленец часто применяется вместе с компоновщиком для представления иерархической структуры в виде графа с совместно используемыми листовыми узлами. Из-за разделения указатель на родителя не может храниться в листовом узле-приспособленце, а должен передаваться ему как часть внешнего состояния. Это оказывает заметное влияние на способ взаимодействия объектов иерархии между собой.

Реализация

При реализации приспособленца следует учитывать следующие аспекты:

- *вынесение внешнего состояния*. Применимость паттерна в значительной степени зависит от того, насколько легко идентифицировать внешнее состояние и вынести его за пределы совместно используемых объектов. Вынесение внешнего состояния не уменьшает затрат на хранение, если различных внешних состояний так же много, как и объектов до совместного использования. Лучший вариант — вычисление внешнего состояния по объектам с другой структурой, требующей значительно меньшей памяти.

  Например, в нашем редакторе документов можно поместить карту с типографской информацией в отдельную структуру, а не хранить шрифт и начертание вместе с каждым символом. В этой карте будут храниться непрерывные серии символов с одинаковыми типографскими атрибутами. Когда объект-символ изображает себя, он получает типографские атрибуты от алгоритма обхода. Поскольку обычно в документах используется немного разных шрифтов и начертаний, то хранить эту информацию

отдельно от объекта-символа гораздо эффективнее, чем непосредственно в нем;

- *управление совместно используемыми объектами*. Так как объекты используются совместно, клиенты не должны создавать экземпляры напрямую. Фабрика `FlyweightFactory` позволяет клиентам найти подходящего приспособленца. В объектах этого класса часто присутствует ассоциативное хранилище, с помощью которого можно быстро находить приспособленца, нужного клиенту. Так, в примере редактора документов фабрика приспособленцев может содержать внутри себя таблицу, индексированную кодом символа, и возвращать нужного приспособленца по его коду. А если требуемый приспособленец отсутствует, он тут же создается.

  Совместное использование также подразумевает некоторую форму подсчета ссылок или уборки мусора для освобождения занимаемой приспособленцем памяти, когда необходимость в нем отпадает. Однако ни то, ни другое необязательно, если число приспособленцев фиксировано и невелико (например, если речь идет о представлении набора символов кода ASCII). В таком случае имеет смысл хранить приспособленцев постоянно.

Пример кода

Возвращаясь к примеру с редактором документов, определим базовый класс `Glyph` для графических объектов-приспособленцев. На логическом уровне глифы — это составные объекты, которые обладают графическими атрибутами и умеют изображать себя (см. описание паттерна компоновщик (196)). Сейчас мы ограничимся только шрифтом, но тот же подход применим и к любым другим графическим атрибутам:

```
class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);

    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

В подклассе `Character` хранится только код символа:

```
class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

Чтобы не выделять память для шрифта в каждом глифе, будем хранить этот атрибут во внешнем объекте класса `GlyphContext`. Данный объект служит хранилищем внешнего состояния. Он поддерживает соответствие между глифом и его шрифтом (а также любыми другими графическими атрибутами) в различных контекстах. Любой операции, у которой должна быть информация о шрифте глифа в данном контексте, в качестве параметра будет передаваться экземпляр `GlyphContext`. У него операция и может запросить нужные сведения. Контекст определяется положением глифа в структуре, поэтому операции обхода и манипулирования потомками должны обновлять `GlyphContext` при каждом использовании:

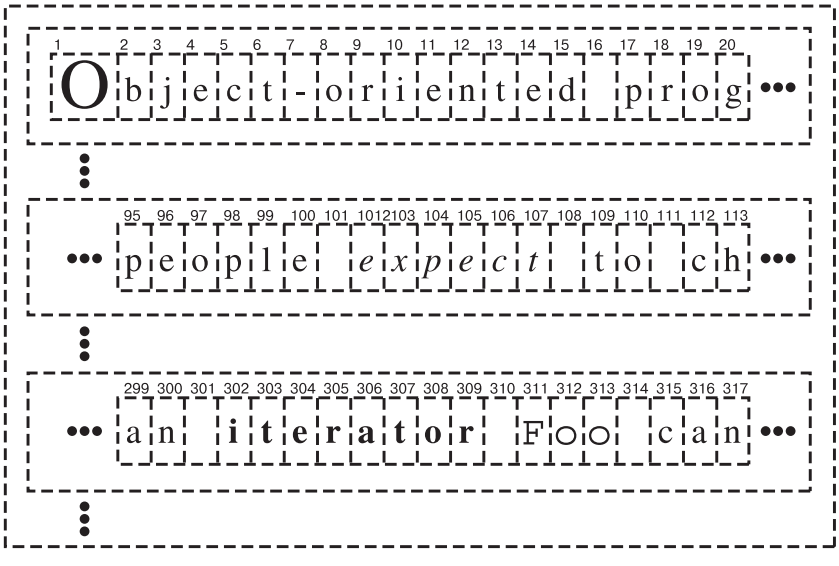
```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```

Объекту `GlyphContext` должно быть известно о текущем положении в структуре глифов во время ее обхода. Операция `GlyphContext::Next` увеличивает переменную `_index` в процессе обхода структуры. Подклассы класса `Glyph`, имеющие потомков (например, `Row` и `Column`), должны реализовывать операцию `Next` так, чтобы она вызывала `GlyphContext::Next` в каждой точке обхода.

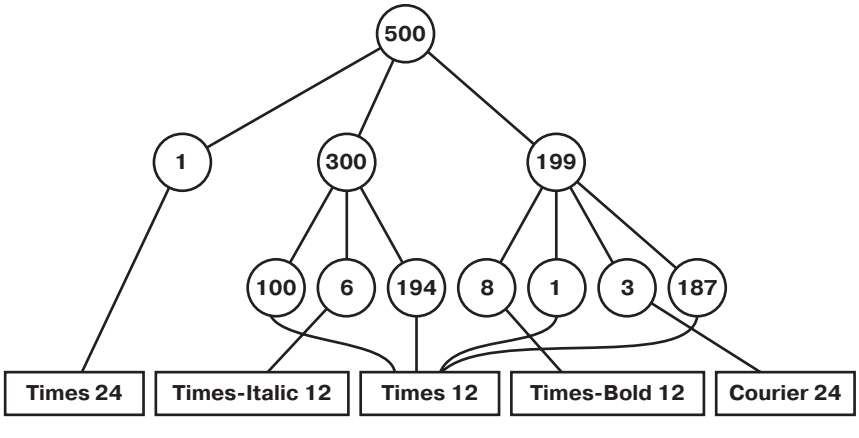
Операция `GlyphContext::GetFont` использует переменную `_index` в качестве ключа для структуры `BTree`, в которой хранится информация соответствия

между глифами и шрифтами. Каждый узел дерева помечен длиной строки, для которой он предоставляет информацию о шрифте. Листья дерева указывают на шрифт, а внутренние узлы разбивают строку на подстроки — по одной для каждого потомка.

Рассмотрим фрагмент текста, представляющий собой композицию глифов.



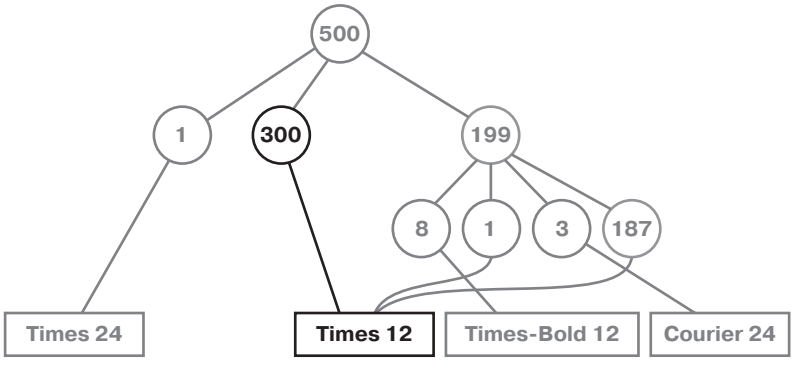
Структура `BTree`, в которой хранится информация о шрифтах, может выглядеть так:



Внутренние узлы определяют диапазоны индексов глифов. Дерево обновляется в ответ на изменение шрифта, а также при каждом добавлении и удалении глифов из структуры. Например, если предположить, что текущей точке обхода соответствует индекс 102, то следующий код установит шрифт каждого символа в слове «expect» таким же, как у близлежащего текста (то есть `times12` — экземпляр класса `Font` для шрифта Times Roman размером 12 пунктов):

```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
// ...

gc.SetFont(times12, 6);
```

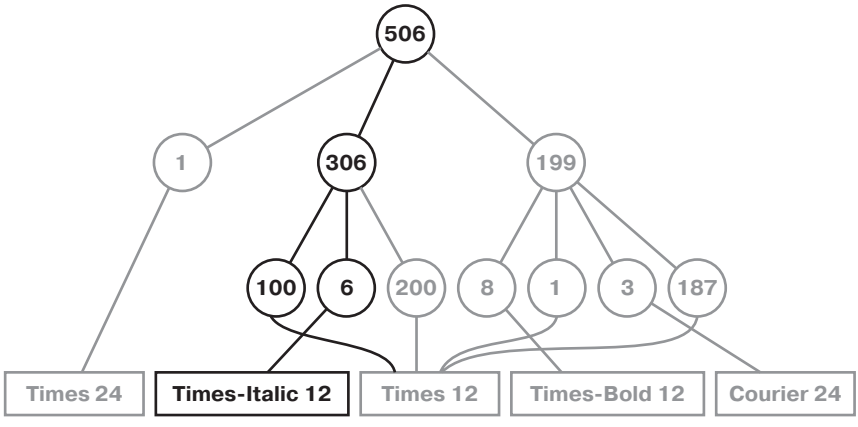
Новая структура `BTree` выглядит так (изменения выделены черным цветом):



Предположим, перед «expect» добавляется слово «don't » (включая пробел после него), написанное шрифтом Times Italic размером 12 пунктов. Следующий код проинформирует объект `gc` об этом (предполагается, что текущей позиции все еще соответствует индекс 102):

```
gc.Insert(6);
gc.SetFont(timesItalic12, 6);
```

Структура `BTree` приходит к следующему виду:



При запросе шрифта текущего глифа объект `GlyphContext` спускается вниз по дереву, суммируя индексы, пока не будет найден шрифт для текущего индекса. Поскольку шрифт меняется нечасто, размер дерева невелик по сравнению с размером структуры глифов. Это позволяет уменьшить затраты на хранение без заметного увеличения времени поиска[1].

Наконец, нужна еще фабрика `FlyweightFactory`, которая создает глифы и обеспечивает их корректное совместное использование. Класс `GlyphFactory` создает объекты `Character` и глифы других видов. Совместно используются только объекты `Character`. Составных глифов гораздо больше, и их существенное состояние (то есть множество потомков) в любом случае является внутренним:

```
const int NCHARCODES = 128;

class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
```

[1] Время поиска в этой схеме пропорционально частоте смены шрифта. Наименьшая производительность достигается, когда шрифт меняется на каждом символе, но на практике это бывает редко.

Массив `_character` содержит указатели на глифы `Character`, индексированные кодом символа. Конструктор инициализирует этот массив нулями:

```
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
         _character[i] = 0;
    }
}
```

Операция `CreateCharacter` ищет символ в массиве и возвращает соответствующий глиф, если он существует. В противном случае `CreateCharacter` создает глиф, помещает его в массив и затем возвращает:

```
Character* GlyphFactory::CreateCharacter (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }

    return _character[c];
}
```

Остальные операции просто создают новый объект при каждом обращении, так как несимвольные глифы не используются совместно:

```
Row* GlyphFactory::CreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}
```

Эти операции можно было бы опустить и позволить клиентам напрямую создавать экземпляры глифов, не используемых совместно. Но если позже вы решите сделать их тоже совместно используемыми, то придется изменять клиентский код, в котором они создаются.

Известные применения

Концепция объектов-приспособленцев впервые была описана и использована как прием проектирования в библиотеке InterViews 3.0 [CL90]. Ее разработчики построили мощный редактор документов Doc, чтобы доказать практическую полезность подобной идеи. В Doc объекты-глифы используются для представления любого символа документа. Редактор строит по одному экземпляру глифа для каждого сочетания символа и стиля (в котором

определены все графические атрибуты). Таким образом, внутреннее состояние символа состоит из его кода и информации о стиле (индекс в таблице стилей)[1]. Следовательно, внешней оказывается только позиция, поэтому Doc работает быстро. Документы представляются классом `Document`, который выполняет функции фабрики `FlyweightFactory`. Измерения показали, что реализованное в Doc совместное использование символов-приспособленцев весьма эффективно. В типичном случае для документа из 180 тысяч знаков достаточно создать только 480 объектов-символов.

В каркасе ET++ [WGM88] приспособленцы используются для поддержки независимого оформления[2]. Его стандарт определяет расположение элементов пользовательского интерфейса (полос прокрутки, кнопок, меню и пр., в совокупности именуемых *виджетами*) и их оформления (тени и т. д.). Виджет делегирует заботу о своем расположении и изображении отдельному объекту `Layout`. Изменение этого объекта позволит сменить оформление даже на стадии выполнения.

Для каждого класса виджета имеется соответствующий класс `Layout` (например, `ScrollbarLayout`, `MenubarLayout` и т. д.). Очевидная проблема такого решения состоит в том, что использование отдельных классов приводит к удвоению числа объектов пользовательского интерфейса, так как для каждого интерфейсного объекта создается дополнительный объект `Layout`. Чтобы избавиться от расходов, объекты `Layout` реализуются в виде приспособленцев. Они прекрасно подходят на эту роль, так как заняты преимущественно определением поведения и им легко передать тот небольшой объем внешней информации о состоянии, необходимый для размещения или прорисовки объекта.

Объекты `Layout` создаются и управляются объектами класса `Look`. Класс `Look` — это абстрактная фабрика (113), которая производит объекты `Layout` с помощью таких операций, как `GetButtonLayout`, `GetMenuBarLayout` и т. д. Для каждого стандарта внешнего облика у класса `Look` есть соответствующий подкласс (`MotifLook`, `OpenLook` и т. д.).

Кстати говоря, объекты `Layout` — это, по существу, стратегии (см. описание паттерна стратегия (362)). Таким образом, мы имеем пример объекта-стратегии, реализованный в виде приспособленца.

---

[1] В приведенном выше примере кода информация о стиле вынесена наружу, так что внутреннее состояние — это только код символа.

[2] Другой подход к обеспечению независимого оформления представлен в описании паттерна абстрактная фабрика.

Родственные паттерны

Паттерн приспособленец часто используется в сочетании с компоновщиком (196) для реализации логической иерархической структуры в виде ациклического направленного графа с совместно используемыми листовыми вершинами.

Часто наилучшим способом реализации объектов состояния (352) и стратегии (362) является паттерн приспособленец.

ПАТТЕРН PROXY (ЗАМЕСТИТЕЛЬ)

Название и классификация паттерна

Заместитель — паттерн, структурирующий объекты.

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Другие названия

`Surrogate` (суррогат).

Мотивация

Одна из причин для управления доступом к объекту — возможность отложить затраты на создание и инициализацию до момента, когда возникнет фактическая необходимость в объекте. Рассмотрим редактор документов, в котором в документы могут встраиваться графические объекты. Затраты на создание некоторых таких объектов (например, больших растровых изображений) могут быть весьма значительными. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты *по требованию*. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как без усложнения реализации редактора скрыть тот факт, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

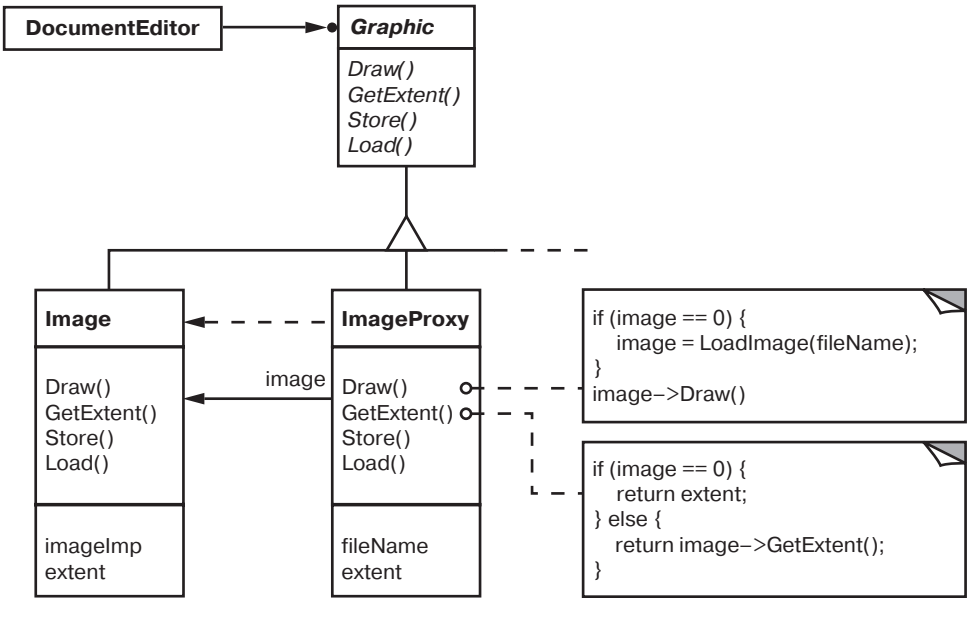
Решение состоит в том, чтобы использовать другой объект — *заместитель* изображения, который временно подставляется вместо реального изобра-

жения. Заместитель ведет себя точно так же, как само изображение, и при необходимости создает его экземпляр.

Заместитель создает настоящее изображение, только если редактор документа вызовет операцию `Draw`. Все последующие запросы заместитель переадресует непосредственно изображению. Поэтому после создания изображения он должен сохранить ссылку на него.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не создавая экземпляр изображения.

На следующей диаграмме классов этот пример показан более подробно.



Редактор документов получает доступ к встроенным изображениям только через интерфейс, определенный в абстрактном классе `Graphic`. `ImageProxy` — это класс для представления изображений, создаваемых по требованию. В `ImageProxy` хранится имя файла, играющее роль ссылки на изображение, которое находится на диске. Имя файла передается конструктору класса `ImageProxy`.

В объекте `ImageProxy` находятся также ограничивающий прямоугольник изображения и ссылка на экземпляр реального объекта `Image`. Ссылка остается недействительной, пока заместитель не создаст экземпляр реального изображения. Операция `Draw` гарантирует, что изображение будет создано до того, как заместитель переадресует ему запрос. Операция `GetExtent` переадресует запрос изображению, только если его экземпляр уже создан; в противном случае `ImageProxy` возвращает те размеры, которые хранит сам.

Применимость

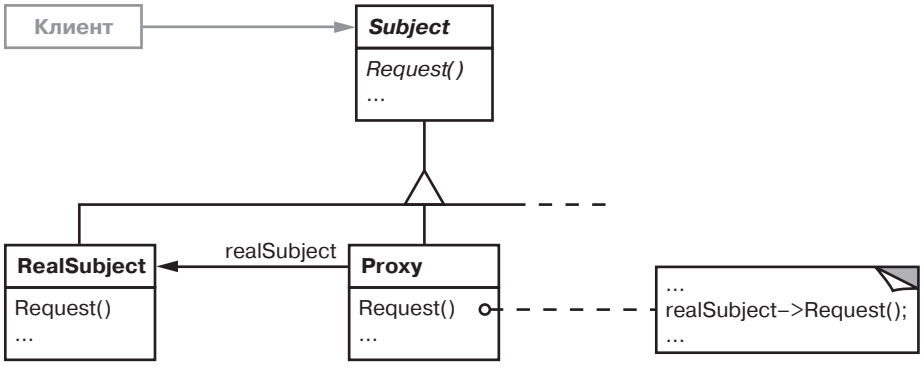
Паттерн заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более гибким или нетривиальным способом, чем при использовании простого указателя. Несколько типичных ситуаций, в которых заместитель может оказаться полезным:

- *удаленный заместитель* предоставляет локального представителя для объекта, находящегося в другом адресном пространстве. В системе NEXTSTEP [Add94] для этой цели применяется класс `NXProxy`. Заместителя такого рода Джеймс Коплиен [Cop92] называет «послом» (Ambassador);
- *виртуальный заместитель* создает «тяжелые» объекты по требованию. Примером может служить класс `ImageProxy`, описанный в разделе «Мотивация»;
- *защищающий заместитель* контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа. Например, в операционной системе Choices [CIRM93] объекты `KernelProxy` ограничивают права доступа к объектам операционной системы;
- *«умная» ссылка* — это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применениям такой ссылки можно отнести:
  - подсчет числа ссылок на реальный объект, с тем чтобы занимаемую им память можно было освободить автоматически, когда не останется

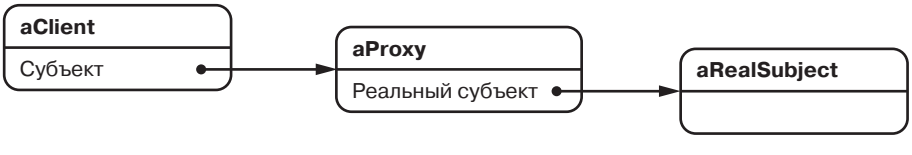
ни одной ссылки (такие ссылки называют еще «умными» указателями [Ede92]);

- загрузку объекта из долгосрочного хранилища в память при первом обращении к нему;
- проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.

Структура



Вот как может выглядеть схема объектов для структуры с заместителем во время выполнения.



Участники

- **Proxy** (ImageProxy) — заместитель:
  - хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса `Proxy` может обращаться к объекту класса `Subject`, если интерфейсы классов `RealSubject` и `Subject` одинаковы;
  - предоставляет интерфейс, идентичный интерфейсу `Subject`, так что заместитель всегда может быть подставлен вместо реального субъекта;
  - контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;

- прочие обязанности зависят от вида заместителя:
  - — *удаленный заместитель* отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;
  - — *виртуальный заместитель* может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание. Например, класс `ImageProxy` из раздела «Мотивация» кэширует размеры реального изображения;
  - — *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

- **Subject** (`Graphic`) — субъект:
  - определяет общий для `RealSubject` и `Proxy` интерфейс, так что класс `Proxy` можно использовать везде, где ожидается `RealSubject`;
- **RealSubject** (`Image`) — реальный субъект:
  - определяет реальный объект, представленный заместителем.

Отношения

`Proxy` при необходимости переадресует запросы объекту `RealSubject`. Детали зависят от вида заместителя.

Результаты

С помощью паттерна заместитель при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

- удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

Есть еще одна оптимизация, которую паттерн заместитель иногда скрывает от клиента. Она называется *копированием при записи* (copy-on-write) и имеет много общего с созданием объекта по требованию. Копирование большого и сложного объекта — очень затратная операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс

копирования, применив паттерн заместитель, то можно быть уверенным, что эта операция произойдет только тогда, когда он действительно был изменен.

Чтобы во время записи можно было копировать, необходимо подсчитывать ссылки на субъект. Копирование заместителя просто увеличивает счетчик ссылок. И только тогда, когда клиент запрашивает операцию, изменяющую субъект, заместитель действительно выполняет копирование. Одновременно заместитель должен уменьшить счетчик ссылок. Когда счетчик ссылок становится равным нулю, субъект уничтожается.

Копирование при записи может существенно уменьшить плату за копирование «тяжелых» субъектов.

Реализация

При реализации паттерна заместитель можно использовать следующие возможности языка:

- *перегрузку оператора обращения к членам класса в C++*. Язык C++ поддерживает перегрузку оператора обращения к членам класса `->`. Это позволяет производить дополнительные действия при любом разыменовании указателя на объект. Для реализации некоторых видов заместителей это оказывается полезно, поскольку заместитель ведет себя аналогично указателю.

  В следующем примере показано, как воспользоваться данным приемом для реализации виртуального заместителя `ImagePtr`:

```
class Image;
extern Image* LoadAnImageFile(const char*);
    // внешняя функция

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
```

```
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

Перегруженные операторы `->` и `*` используют операцию `LoadImage` для возврата клиенту изображения, хранящегося в переменной `_image` (при необходимости загрузив его):

```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

Такой подход позволяет вызывать операции объекта `Image` через объекты `ImagePtr`, не заботясь о том, чтобы включить их в интерфейс данного класса:

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))
```

Обратите внимание, что заместитель изображения ведет себя подобно указателю, но не объявлен как указатель на `Image`. Это означает, что использовать его в точности как настоящий указатель на `Image` нельзя. Поэтому при таком подходе клиентам следует трактовать объекты `Image` и `ImagePtr` по-разному.

Перегрузка оператора доступа не является лучшим решением для всех видов заместителей. Некоторым из них должно быть точно известно, *какая* операция вызывается, а в таких случаях перегрузка оператора доступа не работает.

Возьмем пример виртуального заместителя, обсуждавшийся в разделе «Мотивация». Изображение нужно загружать в точно определенное время — при вызове операции `Draw`, а не при каждом обращении к нему.

Перегрузка оператора доступа не позволяет различить подобные случаи. В такой ситуации придется вручную реализовать каждую операцию заместителя, переадресующую запрос субъекту.

Обычно все эти операции очень похожи друг на друга, как видно из примера кода в одноименном разделе. Они проверяют, что запрос корректен, что объект-адресат существует и т. д., а потом уже перенаправляют ему запрос. Писать этот код снова и снова надоедает, поэтому нередко он автоматически генерируется препроцессором;

- *метод doesNotUnderstand в Smalltalk*. В языке Smalltalk есть возможность, позволяющая автоматически поддерживать переадресацию запросов. При отправке клиентом сообщения, для которого у получателя нет соответствующего метода, Smalltalk вызывает метод `doesNotUnderstand: aMessage`. Заместитель может переопределить `doesNotUnderstand` так, что сообщение будет переадресовано субъекту.

  Чтобы гарантировать, что запрос будет перенаправлен субъекту, а не просто тихо поглощен заместителем, класс `Proxy` можно определить так, что он не станет понимать никаких сообщений. Smalltalk позволяет это сделать, надо лишь, чтобы у `Proxy` не было суперкласса[1].

  Главный недостаток метода `doesNotUnderstand:` в том, что в большинстве Smalltalk-систем имеется несколько специальных сообщений, обрабатываемых непосредственно виртуальной машиной, а в этом случае стандартный механизм поиска методов обходится. Правда, единственной такой операцией, написанной в классе `Object` (а следовательно, способной повлиять на заместителей), является тождество `==`.

  Если вы собираетесь применять `doesNotUnderstand:` для реализации заместителя, вышеописанная проблема должна быть как-то решена на уровне проектирования. Нельзя же ожидать, что совпадение заместителей равнозначно совпадению реальных субъектов. К сожалению, метод `doesNotUnderstand:` изначально создавался для обработки ошибок, а не для построения заместителей, поэтому его быстродействие оставляет желать лучшего;

- *заместителю не всегда должен быть известен тип реального объекта*. Если класс `Proxy` может работать с субъектом только через его абстрактный интерфейс, то не нужно создавать Proxy для каждого класса реаль-

[1] Этот прием используется при реализации распределенных объектов в системе NEXTSTEP [Add94] (точнее, в классе NXProxy). Только там переопределяется метод forward — эквивалент описанного только что приема в Smalltalk.

ного субъекта `RealSubject`; заместитель может обращаться к любому из них единообразно. Но если заместитель должен создавать экземпляры реальных субъектов (как обстоит дело в случае виртуальных заместителей), то знание конкретного класса обязательно.

К проблемам реализации можно отнести и решение вопроса о том, как обращаться к субъекту, экземпляр которого еще не создан. Некоторые заместители должны обращаться к своим субъектам независимо от того, где они находятся — на диске или в памяти. Это означает, что нужно использовать какую-то форму идентификаторов объектов, не зависящих от адресного пространства. В разделе «Мотивация» для этой цели использовалось имя файла.

Пример кода

В следующем коде реализованы два вида заместителей: виртуальный, описанный в разделе «Мотивация», и реализованный с помощью метода `doesNotUnderstand:`[1].

- *виртуальный заместитель*. В классе `Graphic` определен интерфейс для графических объектов:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Класс `Image` реализует интерфейс `Graphic` для отображения графических файлов. В нем замещается операция `HandleMouse`, при помощи которой пользователь может интерактивно изменять размер изображения:

```
class Image : public Graphic {
public:
    Image(const char* file);  // Загружает изображение из файла
    virtual ~Image();
```

[1] Еще один вид заместителя дает паттерн итератор.

```
    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

Класс `ImageProxy` имеет тот же интерфейс, что и `Image`:

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

Конструктор сохраняет локальную копию имени файла, в котором хранится изображение, и инициализирует переменные `_extent` и `_image`:

```
ImageProxy::ImageProxy (const char* fileName)  {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // размеры пока не известны
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

Реализация операции `GetExtent` возвращает кэшированный размер, если это возможно. В противном случае изображение загружается из файла. Операция `Draw` загружает изображение, а `HandleMouse` перенаправляет событие реальному изображению:

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

Операция `Save` записывает кэшированный размер изображения и имя файла в поток, а `Load` считывает эту информацию и инициализирует соответствующие переменные:

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Наконец, предположим, что есть класс `TextDocument` для представления документа, который может содержать объекты класса `Graphic`:

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

    void Insert(Graphic*);
    // ...
};
```

Объект `ImageProxy` можно вставить в документ следующим образом:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

- *заместители, использующие метод doesNotUnderstand.* В языке Smalltalk можно создавать обобщенных заместителей, определяя классы, не имеющие суперкласса[1], а в них — метод `doesNotUnderstand:` для обработки сообщений.

  В показанном ниже фрагменте предполагается, что у заместителя есть метод `realSubject`, возвращающий связанный с ним реальный субъект. При использовании `ImageProxy` этот метод должен был бы проверить, создан ли объект `Image`, при необходимости создать его и затем вернуть. Для обработки перехваченного сообщения, которое было адресовано реальному субъекту, используется метод `perform:withArguments:`.

```
doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
        withArguments: aMessage arguments
```

  Аргументом `doesNotUnderstand:` является экземпляр класса `Message`, представляющий сообщение, не понятое заместителем. Таким образом, при ответе на любое сообщение заместитель сначала проверяет, что реальный субъект существует, а потом уже переадресует ему сообщение.

  Одно из преимуществ метода `doesNotUnderstand:` — возможность выполнения произвольной обработки. Например, можно было бы создать защищающего заместителя, определив набор `legalMessages`-сообщений, которые следует принимать, и передав заместителю следующий метод:

```
doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject
            perform: aMessage selector
            withArguments: aMessage arguments]
        ifFalse: [self error: 'Illegal operator']
```

  Прежде чем переадресовать сообщение реальному субъекту, указанный метод проверяет, что оно действительно. Если это не так, `doesNotUnderstand:` посылает сообщение `error:` самому себе, что приведет к зацикливанию, если в заместителе не определен метод `error:`. Следовательно, определение `error:` должно быть скопировано из класса `Object` вместе со всеми методами, которые в нем используются.

---

[1] Практически для любого класса Object является суперклассом самого верхнего уровня. Поэтому выражение «нет суперкласса» означает то же самое, что «определение класса, для которого Object не является суперклассом».

Известные применения

Пример виртуального заместителя из раздела «Мотивация» заимствован из классов строительного блока текста, определенных в каркасе ET++.

В системе NEXTSTEP [Add94] заместители (экземпляры класса NXProxy) используются как локальные представители объектов, которые могут быть распределенными. Сервер создает заместителей для удаленных объектов, когда клиент их запрашивает. Заместитель кодирует полученное сообщение вместе со всеми аргументами, после чего отправляет его удаленному субъекту. Аналогично субъект кодирует возвращенные результаты и посылает их обратно объекту `NXProxy`.

В работе McCullough [McC87] обсуждается применение заместителей в Smalltalk для обращения к удаленным объектам. Джефри Пэско (Geoffrey Pascoe) [Pas86] описывает, как обеспечить побочные эффекты при вызове методов и реализовать контроль доступа с помощью «инкапсуляторов».

Родственные паттерны

Адаптер (171): предоставляет другой интерфейс к адаптируемому объекту. Напротив, заместитель в точности повторяет интерфейс своего субъекта. Однако, если заместитель используется для ограничения доступа, он может отказаться выполнять операцию, которую субъект выполнил бы, поэтому на самом деле интерфейс заместителя может быть и подмножеством интерфейса субъекта.

Декоратор (209): хотя его реализация и похожа на реализацию заместителя, но назначение совершенно иное. Декоратор добавляет объекту новые обязанности, а заместитель контролирует доступ к объекту.

Степень схожести реализации заместителей и декораторов может быть различной. Защищающий заместитель мог бы быть реализован в точности как декоратор. С другой стороны, удаленный заместитель не содержит прямых ссылок на реальный субъект, а лишь косвенную ссылку, что-то вроде «идентификатор хоста и локальный адрес на этом хосте». Вначале виртуальный заместитель имеет только косвенную ссылку (скажем, имя файла), но в конечном итоге получает и использует прямую ссылку.

ОБСУЖДЕНИЕ СТРУКТУРНЫХ ПАТТЕРНОВ

Возможно, вы обратили внимание на некоторое сходство между структурными паттернами, особенно в том, что касается их участников и взаимо-

действий. Скорее всего, это объясняется тем, что все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (одиночном и множественном наследовании для паттернов уровня класса и композиции для паттернов уровня объектов). Тем не менее, сходство может быть обманчиво, так как с помощью разных паттернов можно решать совершенно разные задачи. В этом разделе сопоставлены группы структурных паттернов, и вы сможете яснее представить их сравнительные достоинства и недостатки.

АДАПТЕР И МОСТ

У паттернов адаптер (171) и мост (184) есть несколько общих атрибутов. Тот и другой повышают гибкость, вводя дополнительный уровень косвенности при обращении к другому объекту. Оба перенаправляют запросы другому объекту, используя иной интерфейс.

Принципиальное различие между адаптером и мостом в их назначении. Цель первого — устранение несовместимости между двумя существующими интерфейсами. При разработке адаптера не учитывается, как эти интерфейсы реализованы и то, как они могут независимо развиваться в будущем. Он должен лишь обеспечить совместную работу двух независимо разработанных классов, так чтобы ни один из них не пришлось переделывать. С другой стороны, мост связывает абстракцию с ее, возможно, многочисленными реализациями. Данный паттерн предоставляет клиентам стабильный интерфейс, позволяя в то же время изменять классы, которые его реализуют. Мост также подстраивается под новые реализации, появляющиеся в процессе развития системы.

В связи с этими различиями адаптер и мост часто используются в разные моменты жизненного цикла системы. Когда выясняется, что два несовместимых класса должны работать вместе, часто приходится пользоваться адаптером — обычно для того, чтобы избежать дублирования кода. Заранее такую ситуацию предвидеть нельзя. Наоборот, пользователь моста с самого начала понимает, что у абстракции может быть несколько реализаций и развитие того и другого будет идти независимо. Адаптер обеспечивает работу *после* того, как нечто спроектировано; мост — *до* того. Это доказывает, что адаптер и мост предназначены для решения именно своих задач.

Фасад (221) можно рассматривать как адаптер к набору других объектов. Но такая интерпретация упускает один нюанс: фасад определяет *новый* интерфейс, тогда как адаптер повторно использует уже имеющийся. Вспомните,

что адаптер заставляет работать вместе два *существующих* интерфейса, а не определяет новый.

КОМПОНОВЩИК, ДЕКОРАТОР И ЗАМЕСТИТЕЛЬ

У компоновщика (196) и декоратора (209) похожие структурные схемы, что указывает на то, что оба паттерна основаны на рекурсивной композиции и предназначены для организации заранее неопределенного числа объектов. При обнаружении данного сходства может возникнуть искушение посчитать объект-декоратор вырожденным случаем компоновщика, но при этом будет искажен сам смысл паттерна декоратор. Сходство и заканчивается на рекурсивной композиции, и снова из-за различия задач, решаемых с помощью паттернов.

Назначение декоратора — добавить новые обязанности объекта без порождения подклассов. Этот паттерн позволяет избежать комбинаторного роста числа подклассов, если проектировщик пытается статически определить все возможные комбинации. У компоновщика другие задачи. Он должен так структурировать классы, чтобы с разными взаимосвязанными объектами можно было бы работать единообразно, а несколько объектов обрабатывать как один. Акцент здесь делается не на оформлении, а на представлении.

Эти цели различны, но они дополняют друг друга, поэтому компоновщик и декоратор часто используются совместно. Оба паттерна позволяют спроектировать систему так, что приложения можно будет создавать, просто соединяя объекты между собой, без определения новых классов. Появится некий абстрактный класс, одни подклассы которого — компоновщики, другие — декораторы, а третьи — реализации фундаментальных строительных блоков системы. В таком случае у компоновщиков и декораторов будет общий интерфейс. С точки зрения паттерна декоратор компоновщик является конкретным компонентом. С точки зрения компоновщика декоратор — это листовый узел. Разумеется, их *не обязательно* использовать вместе, и, как было показано выше, эти паттерны имеют разные цели.

Заместитель (246) — еще один паттерн, структура которого напоминает декоратор. Оба паттерна описывают формирование уровня косвенного доступа к объекту, а в реализации объектов-декораторов и заместителей хранится ссылка на другой объект, которому переадресуются запросы. Но и здесь цели различаются.

Как и декоратор, паттерн заместитель предоставляет клиенту интерфейс, совпадающий с интерфейсом замещаемого объекта. Но в отличие от декоратора

заместителю не нужно динамически добавлять и отбирать свойства, он не предназначен для рекурсивной композиции. Заместитель должен предоставить стандартную замену субъекту, когда прямой доступ к нему неудобен или нежелателен, например потому, что он находится на удаленной машине, хранится на диске или доступен лишь ограниченному кругу клиентов.

В паттерне заместитель субъект определяет ключевую функциональность, а заместитель разрешает (или запрещает) доступ к ней. В декораторе компонент предоставляет лишь часть функциональности, а остальное привносят один или несколько декораторов. Декоратор предназначен для ситуаций, в которых полную функциональность объекта нельзя определить на этапе компиляции или это по крайней мере неудобно. Такая неопределенность делает рекурсивную композицию неотъемлемой частью декоратора. В случае с заместителем дело обстоит не так, ибо ему важно лишь одно отношение — между собой и своим субъектом, а данное отношение можно выразить статически.

Указанные различия существенны, поскольку в них абстрагированы решения конкретных проблем, снова и снова возникающих при объектно-ориентированном проектировании. Но это не означает, что паттерны не могут комбинироваться. Можно представить себе заместителя-декоратора, который добавляет новую функциональность заместителю, или декоратора-заместителя, который оформляет удаленный объект. Такие гибриды теоретически *могут* быть полезны (у нас, правда, не нашлось реального примера), а вот паттерны, из которых они составлены, полезны наверняка.

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

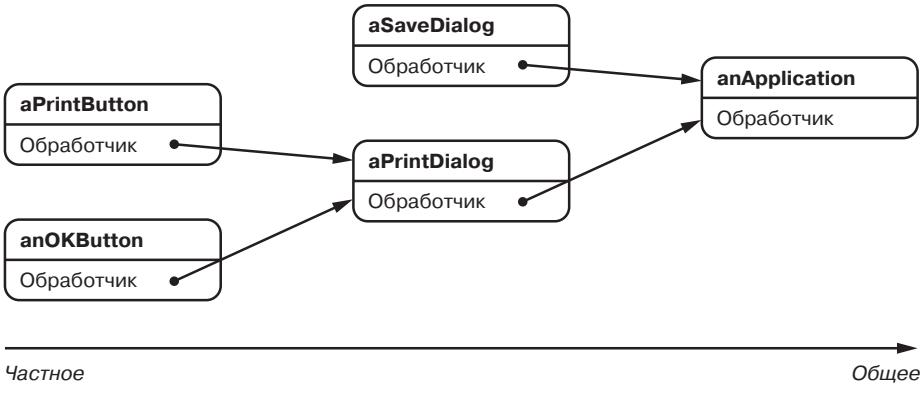
Рассмотрим контекстнозависимую оперативную справку в графическом интерфейсе: пользователь может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание

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.

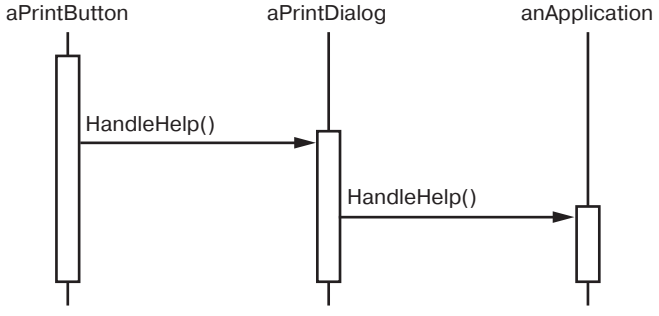
Идея паттерна заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока не будет обработан одним из них.



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).

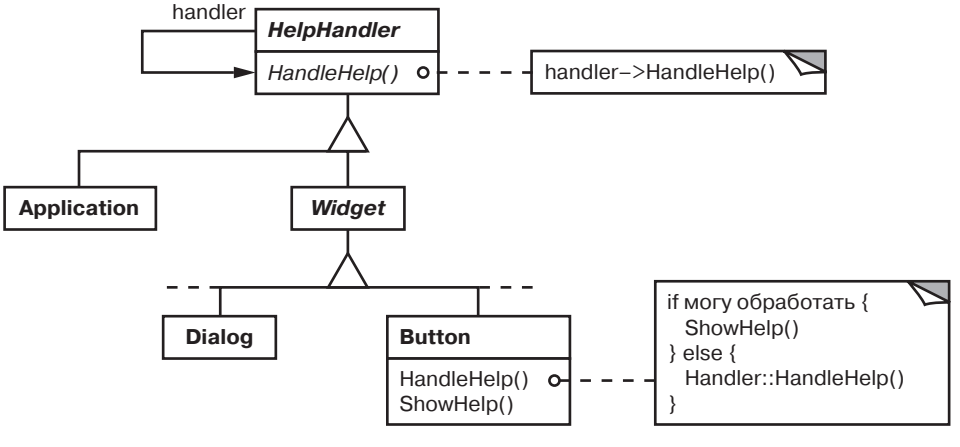
Допустим, пользователь запрашивает справку по кнопке Print (печать). Она находится в диалоговом окне `PrintDialog`, содержащем информацию об объ-

екте приложения, которому принадлежит (см. предыдущую диаграмму). На следующей диаграмме взаимодействий показано, как запрос на получение справки перемещается по цепочке.



В данном случае ни кнопка `aPrintButton`, ни окно `aPrintDialog` не обрабатывают запрос; он достигает объекта `anApplication`, который может его обработать или игнорировать. У клиента, инициировавшего запрос, нет прямой ссылки на объект, который его в конце концов выполнит.

Чтобы отправить запрос по цепочке и гарантировать анонимность получателя, все объекты в цепочке имеют единый интерфейс для обработки запросов и для доступа к своему *преемнику* (следующему объекту в цепочке). Например, в системе оперативной справки можно было бы определить класс `HelpHandler` (предок классов всех объектов-кандидатов или класс-примесь (mixin class)) с операцией `HandleHelp`. Тогда классы, которые хотят обрабатывать запросы, могут сделать `HelpHandler` своим родителем:



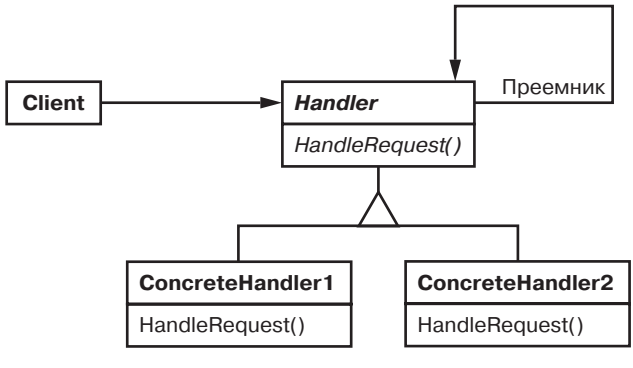
Для обработки запросов на получение справки классы `Button`, `Dialog` и `Application` пользуются операциями `HelpHandler`. По умолчанию операция `HandleHelp` просто перенаправляет запрос своему преемнику. В подклассах эта операция замещается, так что при благоприятных обстоятельствах может выдаваться справочная информация. В противном случае запрос отправляется дальше посредством реализации по умолчанию.

Применимость

Основные условия для применения паттерна цепочка обязанностей:

- запрос может быть обработан более чем одним объектом, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- запрос должен быть отправлен одному из нескольких объектов, без явного указания, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Структура



Типичная структура объектов выглядит примерно так:

Участники

- **Handler** (`HelpHandler`) — обработчик:
  - определяет интерфейс для обработки запросов;
  - (необязательно) реализует связь с преемником;
- **ConcreteHandler** (`PrintButton`, `PrintDialog`) — конкретный обработчик:
  - обрабатывает запрос, за который отвечает;
  - имеет доступ к своему преемнику;
  - если `ConcreteHandler` способен обработать запрос, то так и делает, если не может, то направляет его своему преемнику;
- **Client** — клиент:
  - отправляет запрос некоторому объекту `ConcreteHandler` в цепочке.

Отношения

Запрос, инициированный клиентом, продвигается по цепочке, пока некоторый объект `ConcreteHandler` не возьмет на себя ответственность за его обработку.

Результаты

Основные достоинства и недостатки паттерна цепочка обязанностей:

- *ослабление связанности*. Паттерн освобождает объект от необходимости знать, кто конкретно обработает его запрос. Отправитель и получатель ничего не знают друг о друге, а включенный в цепочку объект — о структуре цепочки.

  В результате цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо хранения ссылок на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;

- *дополнительная гибкость при распределении обязанностей между объектами*. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков;

- *получение не гарантировано*. Поскольку у запроса нет явного получателя, то *нет и гарантий*, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Реализация

При рассмотрении цепочки обязанностей следует обратить внимание на следующие аспекты:

- *реализация цепочки преемников*. Такую цепочку можно реализовать двумя способами:
  - определить новые связи (обычно это делается в классе `Handler`, но можно и в `ConcreteHandler`);
  - использовать существующие связи.

  До сих пор в наших примерах определялись новые связи, однако можно воспользоваться уже имеющимися ссылками на объекты для формирования цепочки преемников. Например, ссылка на родителя в иерархии «часть — целое» может заодно определять и преемника «части». В структуре виджетов такие связи тоже могут существовать. В разделе, посвященном паттерну компоновщик (196), ссылки на родителей обсуждаются более подробно.

  Существующие связи хорошо работают тогда, когда они уже поддерживают нужную цепочку. Это позволит избежать явного определения новых связей и сэкономить память. Но если структура не отражает устройства цепочки обязанностей, то уйти от определения избыточных связей не удастся;

- *соединение преемников*. Если готовых ссылок, которые могли бы использоваться для определения цепочки, нет, то их придется ввести. В таком случае класс `Handler` не только определяет интерфейс запросов, но еще и хранит ссылку на преемника. Следовательно у обработчика появляется возможность определить реализацию операции `HandleRequest` по умолчанию — перенаправление запроса преемнику (если таковой существует). Если запрос не представляет интереса для подкласса `ConcreteHandler`, то последнему не нужно замещать эту операцию, поскольку по умолчанию запрос будет отправлен дальше.

  Пример базового класса `HelpHandler`, в котором хранится указатель на преемника:

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

- *представление запросов.* Представлять запросы можно по-разному. В простейшей форме (как в классе `HandleHelp`) запрос жестко кодируется в виде вызова некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе `Handler`.

  Альтернатива — использование одной функции-обработчика, которой передается код запроса (скажем, целое число или строка). Так можно поддержать заранее неизвестное число запросов. Единственное требование состоит в том, что отправитель и получатель должны договориться о способе кодирования запроса.

  Этот подход более гибок, но при реализации нужно использовать условные операторы для передачи запросов в зависимости от их кодов. Кроме того, не существует безопасного с точки зрения типов способа передачи параметров, поэтому упаковывать и распаковывать их приходится вручную. Очевидно, что это не так безопасно, как прямой вызов операции.

  Чтобы решить проблему передачи параметров, допустимо использовать отдельные *объекты-запросы*, в которых инкапсулируются параметры запроса. Класс `Request` может представлять некоторые запросы явно, а их новые типы описываются в подклассах. Подкласс может определить другие параметры. Чтобы получить доступ к этим параметрам, обработчик должен располагать информацией о типе запроса (какой именно подкласс `Request` используется).

  Для идентификации запроса в классе `Request` можно определить функцию доступа, которая возвращает идентификатор класса. Вместо этого получатель мог бы воспользоваться информацией о типе, доступной во время выполнения, если язык программирования поддерживает такую возможность.

Приведем пример функции диспетчеризации, в которой используются объекты для идентификации запросов. Операция `GetKind`, указанная в базовом классе `Request`, определяет вид запроса:

```
void Handler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
    case Help:
        // привести аргумент к подходящему типу
        HandleHelp((HelpRequest*) theRequest);
        break;

    case Print:
        HandlePrint((PrintRequest*) theRequest);
        // ...
        break;

    default:
        // ...
        break;
    }
}
```

Подклассы могут расширить схему диспетчеризации, переопределив операцию `HandleRequest`. Подкласс обрабатывает лишь те запросы, в которых заинтересован, а остальные отправляет родительскому классу. В этом случае подкласс именно расширяет, а не замещает операцию `HandleRequest`. Например, подкласс `ExtendedHandler` расширяет операцию `HandleRequest`, определенную в классе `Handler`, следующим образом:

```
class ExtendedHandler : public Handler {
public:
    virtual void HandleRequest(Request* theRequest);
    // ...
};

void ExtendedHandler::HandleRequest(Request*theRequest){
    switch (theRequest->GetKind()) {
    case Preview:
        // Обработать запрос Preview
        break;
        default:
        // Дать классу Handler возможность
        // обработать остальные запросы
        Handler::HandleRequest(theRequest);
    }
}
```

- *автоматическое перенаправление запросов в языке Smalltalk.* С этой целью можно использовать механизм `doesNotUnderstand`. Сообщения, не имеющие соответствующих методов, перехватываются реализацией `doesNotUnderstand`, которая может быть замещена для перенаправления сообщения объекту-преемнику. Таким образом, осуществлять перенаправление вручную необязательно. Класс обрабатывает только те запросы, в которых он заинтересован, и рассчитывает, что механизм `doesNotUnderstand` перенаправит все остальные.

Пример кода

Следующий пример показывает, как с помощью цепочки обязанностей можно обработать запросы к описанной выше системе оперативной справки. Запрос на получение справки — это явная операция. Мы воспользуемся уже имеющимися в иерархии виджетов ссылками для перемещения запросов по цепочке от одного виджета к другому и определим в классе `Handler` отдельную ссылку, чтобы можно было передать запрос включенным в цепочку объектам, не являющимся виджетами.

Класс `HelpHandler` определяет интерфейс для обработки запросов на получение справки. В нем хранится раздел справки (по умолчанию пустой) и ссылка на преемника в цепочке обработчиков. Основной операцией является `HandleHelp`, которая замещается в подклассах. `HasHelp` — это вспомогательная операция, проверяющая, ассоциирован ли с объектом какой-нибудь раздел:

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}
```

```
void HelpHandler::HandleHelp () {
    if (_successor != 0) {
    _successor->HandleHelp();
    }
}
```

Все виджеты — подклассы абстрактного класса `Widget`, который, в свою очередь, является подклассом `HelpHandler`, так как со всеми элементами пользовательского интерфейса может быть ассоциирована справочная информация. (Конечно, можно было построить реализацию и на основе класса-примеси.)

```
class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}
```

В нашем примере первым обработчиком в цепочке является кнопка. Класс `Button` — это подкласс `Widget`. Конструктор класса `Button` получает два параметра — ссылку на виджет, в котором он находится, и раздел справки:

```
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // Операции класса Widget, которые Button замещает...
};
```

Реализация `HandleHelp` из класса `Button` сначала проверяет, есть ли для кнопки справочная информация. Если разработчик не определил ее, то запрос отправляется преемнику с помощью операции `HandleHelp` класса `HelpHandler`. Если же информация *есть*, то кнопка ее выводит, и поиск заканчивается:

```
Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // Предоставить справку по кнопке
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Класс `Dialog` реализует аналогичную схему, только его преемником является не виджет, а *произвольный* обработчик запроса на справку. В нашем приложении таким преемником выступает экземпляр класса `Application`:

```
class Dialog : public Widget {
public:
    Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // Операции класса Widget, которые Dialog замещает...
    // ...
};

Dialog::Dialog (HelpHandler* h,  Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // Предоставить справку по диалоговому окну
    } else {
        HelpHandler::HandleHelp();
    }
}
```

В конце цепочки находится экземпляр класса `Application`. Приложение — это не виджет, поэтому `Application` — прямой потомок класса `HelpHandler`. Если запрос на получение справки дойдет до этого уровня, то класс `Application` может выдать информацию о приложении в целом или предложить список разделов справки:

```
class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }
    virtual void HandleHelp();
    // Операции, относящиеся к самому приложению...
};

void Application::HandleHelp () {
    // Показать список разделов справки
}
```

Следующий код создает и связывает эти объекты. В данном случае рассматривается диалоговое окно Print, поэтому с объектами связаны разделы справки, относящиеся к печати:

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;
```

```
Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

Мы можем инициировать запрос на получение справки, вызвав операцию `HandleHelp` для любого объекта в цепочке. Чтобы начать поиск с объекта кнопки, достаточно выполнить его операцию `HandleHelp`:

```
button->HandleHelp();
```

В этом примере кнопка обрабатывает запрос сразу же. Заметим, что класс `HelpHandler` можно было бы сделать преемником `Dialog`. Более того, его преемника можно изменять динамически. Вот почему, где бы диалоговое окно ни встретилось, вы всегда получите справочную информацию с учетом контекста.

Известные применения

Паттерн цепочка обязанностей используется в нескольких библиотеках классов для обработки событий, инициированных пользователем. Класс `Handler` в них называется по-разному, но идея всегда одна и та же: когда пользователь щелкает кнопкой мыши или нажимает клавишу, генерируется некоторое событие, которое распространяется по цепочке. В MacApp [App89] и ET++ [WGM88] класс называется `EventHandler`, в библиотеке TCL фирмы Symantec [Sym93b] — `Bureaucrat`, а в библиотеке из системы NeXT [Add94] используется имя `Responder`.

В каркасе графических редакторов Unidraw определены объекты `Command`, которые инкапсулируют запросы к объектам `Component` и `ComponentView` [VL90]. Объекты `Command` являются запросами в том смысле, что они могут интерпретироваться компонентом или представлением компонента как команда на выполнение определенной операции. Это соответствует подходу «запрос как объект», описанному в разделе «Реализация». Компоненты и виды компонентов могут быть организованы иерархически. Как компонент, так и его представление могут перепоручать интерпретацию команды своему родителю, тот — своему родителю и так далее, то есть речь идет о типичной цепочке обязанностей.

В ET++ паттерн цепочка обязанностей применяется для обработки запросов на обновление графического изображения. Графический объект вызывает операцию `InvalidateRect` всякий раз, когда возникает необходимость обновить часть занимаемой им области. Но выполнить эту операцию самостоятельно графический объект не может, так как не имеет достаточной

информации о своем контексте, например из-за того, что окружен такими объектами, как `Scroller` (полоса прокрутки) или `Zoomer` (лупа), которые преобразуют его систему координат. Это означает, что объект может быть частично невидим, так как он оказался за границей области прокрутки или изменился его масштаб. Поэтому реализация `InvalidateRect` по умолчанию переадресует запрос контейнеру, где находится соответствующий объект. Последний объект в цепочке обязанностей — экземпляр класса `Window`. К тому моменту, когда `Window` получит запрос, недействительный прямоугольник будет гарантированно преобразован правильно. `Window` обрабатывает `InvalidateRect`, послав запрос интерфейсу оконной системы и требуя тем самым выполнить обновление.

Родственные паттерны

Паттерн цепочка обязанностей часто применяется вместе с паттерном компоновщик (196). В этом случае родитель компонента может выступать в роли его преемника.

ПАТТЕРН COMMAND (КОМАНДА)

Название и классификация паттерна

Команда — паттерн поведения объектов.

Назначение

Инкапсулирует запрос в объекте, позволяя тем самым параметризовать клиенты для разных запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Другие названия

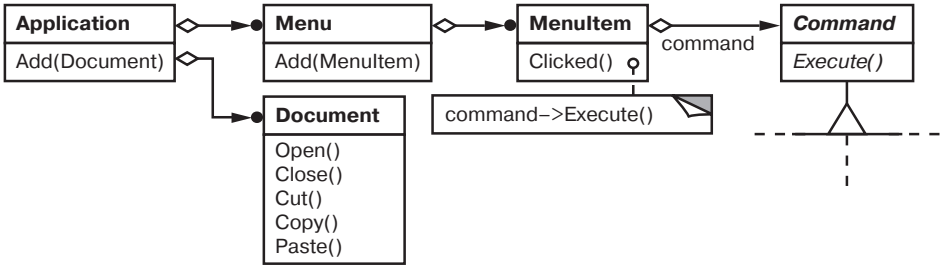
`Action` (действие), `Transaction` (транзакция).

Мотивация

Иногда необходимо посылать объектам запросы, ничего не зная о том, выполнение какой операции запрошено и кто является получателем. Например, в библиотеках для построения пользовательских интерфейсов встречаются такие объекты, как кнопки и меню, которые посылают запрос в ответ на действие пользователя. Но сама библиотека не может явно реализовать запрос в кнопке или меню, потому что только приложение, использующее

библиотеку, располагает информацией о том, что следует сделать. Проектировщик библиотеки ничего не знает о получателе запроса и о том, какие операции тот должен выполнить.

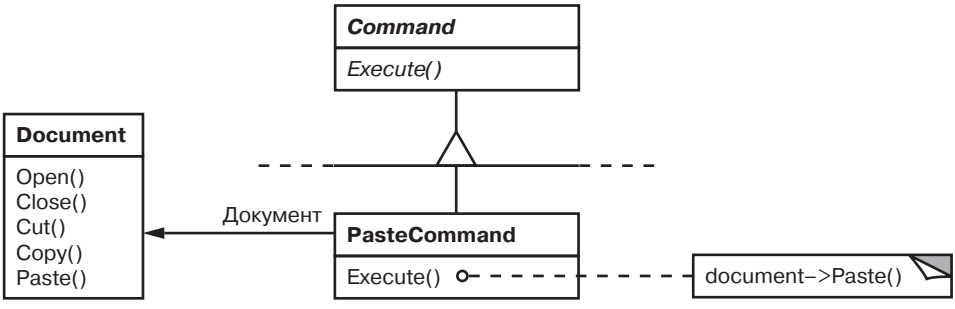
Паттерн команда позволяет библиотечным объектам отправлять запросы неизвестным объектам приложения, преобразовав сам запрос в объект. Этот объект можно хранить и передавать, как и любой другой. В основе списываемого паттерна лежит абстрактный класс `Command`, в котором объявлен интерфейс для выполнения операций. В простейшей своей форме этот интерфейс состоит из одной абстрактной операции Execute. Конкретные подклассы `Command` определяют пару «получатель — действие», сохраняя получателя в переменной экземпляра, и реализуют операцию `Execute`, так чтобы она посылала запрос. У получателя есть информация, необходимая для выполнения запроса.



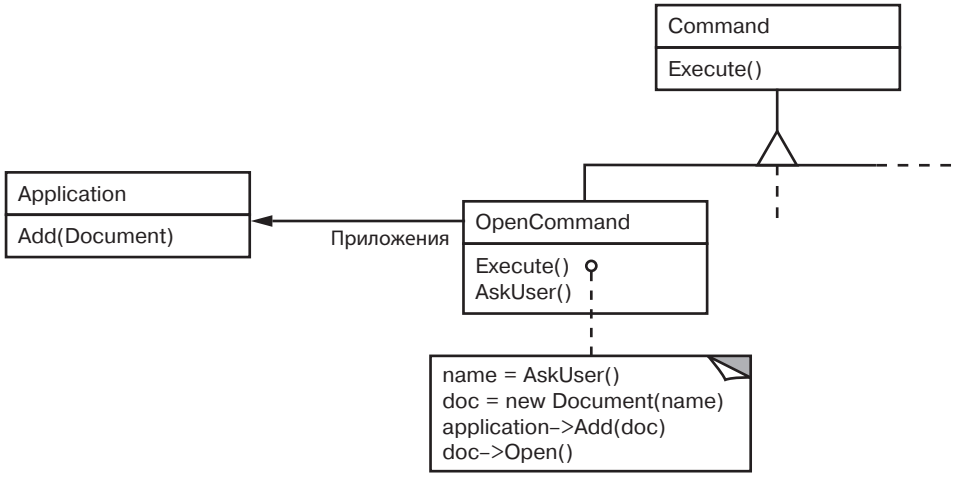
Меню легко реализуются с помощью объектов `Command`. Каждый пункт меню — это экземпляр класса `MenuItem`. Сами меню и все их пункты создает класс `Application` наряду со всеми остальными элементами пользовательского интерфейса. Класс `Application` отслеживает также открытые пользователем документы.

Приложение конфигурирует каждый объект `MenuItem` экземпляром конкретного подкласса `Command`. Когда пользователь выбирает некоторый пункт меню, ассоциированный с ним объект `MenuItem` вызывает `Execute` для своего объекта-команды, а `Execute` выполняет операцию. Объекты `MenuItem` не имеют информации, какой подкласс класса `Command` они используют. Подклассы `Command` хранят информацию о получателе запроса и вызывают одну или несколько операций этого получателя.

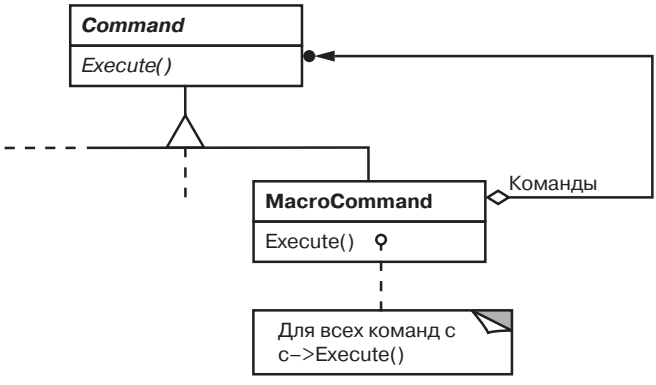
Например, подкласс `PasteCommand` поддерживает вставку текста из буфера обмена в документ. Получателем для `PasteCommand` является объект `Document`, который был передан при создании объекта. Операция `Execute` вызывает операцию `Paste` документа-получателя.



Для подкласса `OpenCommand` операция `Execute` ведет себя по-другому: она запрашивает у пользователя имя документа, создает соответствующий объект `Document`, оповещает о новом документе приложение-получатель и открывает этот документ.



Иногда объект `MenuItem` должен выполнить *последовательность* команд. Например, пункт меню для центрирования страницы стандартного размера можно было бы сконструировать сразу из двух объектов: `CenterDocumentCommand` и `NormalSizeCommand`. Поскольку такое комбинирование команд — явление обычное, то мы можем определить класс `MacroCommand`, позволяющий объекту `MenuItem` выполнять произвольное число команд. `MacroCommand` — это конкретный подкласс класса `Command`, который просто выполняет последовательность команд. У него нет явного получателя, поскольку для каждой команды определен свой собственный.



Обратите внимание: в каждом из этих примеров паттерн команда отделяет объект, инициирующий операцию, от объекта, который располагает информацией, необходимой для ее выполнения. Это позволяет добиться высокой гибкости при проектировании пользовательского интерфейса. Пункт меню и кнопка одновременно могут быть ассоциированы в приложении с некоторой функцией, для этого достаточно приписать обоим элементам один и тот же экземпляр конкретного подкласса класса `Command`. Команды могут заменяться динамически, что очень полезно для реализации контекстно-зависимых меню. Можно также поддержать сценарии, если компоновать простые команды в более сложные. Все это выполнимо потому, что объект, инициирующий запрос, должен располагать информацией лишь о том, как его отправить, а не о том, как он должен выполняться.

Применимость

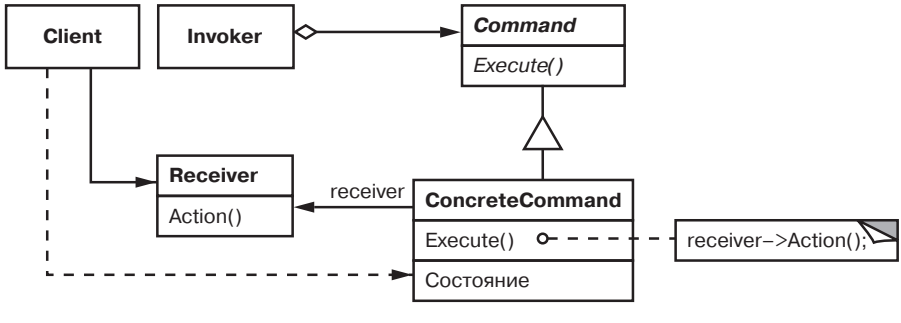
Основные условия для применения паттерна команда:

- *параметризация объектов выполняемым действием, как в случае с пунктами меню MenuItem*. В процедурном языке такую параметризацию можно выразить с помощью *функции обратного вызова*, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представляют собой объектно-ориентированную альтернативу функциям обратного вызова;
- *определение, постановка в очередь и выполнение запросов в разное время*. Время жизни объекта `Command` не обязательно должно зависеть от времени жизни исходного запроса. Если получатель запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;
- *поддержка отмены операций*. Операция `Execute` объекта `Command` может сохранить состояние, необходимое для отмены действий, выполненных

командой. В этом случае в интерфейсе класса `Command` должна быть дополнительная операция `Unexecute`, которая отменяет действия, выполненные предшествующим обращением к `Execute`. Выполненные команды хранятся в списке истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и прямом направлениях, вызывая при посещении каждого элемента команду `Unexecute` или `Execute`;

- *поддержка протоколирования изменений, чтобы их можно было выполнить повторно после сбоя в системе.* Дополнив интерфейс класса `Command` операциями сохранения и загрузки, вы сможете вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции `Execute`;
- *структурирование системы на основе высокоуровневых операций, построенных из примитивных.* Такая структура типична для информационных систем с поддержкой транзакций. Транзакция инкапсулирует набор изменений данных. Паттерн команда позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Структура



Участники

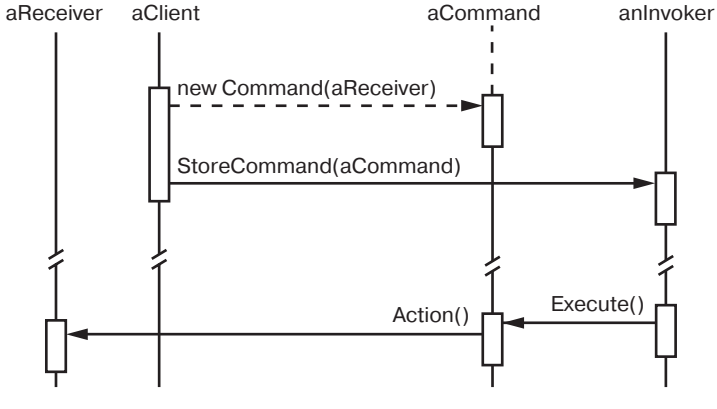
- **Command** — команда:
  - объявляет интерфейс для выполнения операции;
- **ConcreteCommand** (`PasteCommand`, `OpenCommand`) — конкретная команда:
  - определяет связь между объектом-получателем `Receiver` и действием;
  - реализует операцию `Execute` путем вызова соответствующих операций объекта `Receiver`;

- **Client** (`Application`) — клиент:
  - создает объект класса `ConcreteCommand` и устанавливает его получателя;
- **Invoker** (`MenuItem`) — инициатор:
  - обращается к команде для выполнения запроса;
- **Receiver** (`Document`, `Application`) — получатель:
  - располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

Отношения

- клиент создает объект `ConcreteCommand` и устанавливает для него получателя;
- инициатор `Invoker` сохраняет объект `ConcreteCommand`;
- инициатор отправляет запрос, вызывая операцию команды `Execute`. Если поддерживается отмена выполненных действий, то `ConcreteCommand` перед вызовом `Execute` сохраняет информацию о состоянии, достаточную для выполнения отмены;
- объект `ConcreteCommand` вызывает операции получателя для выполнения запроса.

На следующей схеме видно, как `Command` разрывает связь между инициатором и получателем (а также запросом, который должен выполнить последний).



Результаты

Основные результаты применения паттерна команда:

- команда отделяет объект, инициирующий операцию, от объекта, располагающего информацией о том, как ее выполнить;
- команды — это самые настоящие объекты. Их можно обрабатывать и расширять точно так же, как любые другие объекты;
- из простых команд можно собирать составные, например класс `MacroCommand`, рассмотренный выше. В общем случае составные команды описываются паттерном компоновщик (196);
- новые команды добавляются легко, поскольку никакие существующие классы изменять не нужно.

Реализация

При реализации паттерна команда следует обратить внимание на следующие аспекты:

- *насколько «умной» должна быть команда*. У команды может быть широкий круг обязанностей, от простого определения связи между получателем и действиями, которые нужно выполнить для удовлетворения запроса, до самостоятельной реализации без обращения за помощью к получателю. Последний вариант полезен, когда вы хотите определить команды, не зависящие от существующих классов, когда подходящего получателя не существует или когда получатель команде точно не известен. Например, команда, создающая новое окно приложения, может не понимать, что именно она создает, а трактовать окно как любой другой объект. Где-то посередине между двумя крайностями находятся команды, обладающие достаточной информацией для динамического обнаружения своего получателя;
- *поддержка отмены и повтора операций*. Команды могут поддерживать отмену и повтор операций, если имеется возможность отменить результаты выполнения (например, операции `Unexecute` или `Undo`). В классе `ConcreteCommand` может сохраняться необходимая для этого дополнительная информация, в том числе:
  - объект-получатель `Receiver`, который выполняет операции в ответ на запрос;
  - аргументы операции, выполненной получателем;

- исходные значения различных атрибутов получателя, которые могли измениться в результате обработки запроса. Получатель должен предоставить операции, позволяющие команде вернуться в исходное состояние.

Для поддержки всего одного уровня отмены приложению достаточно сохранять только последнюю выполненную команду. Если же нужны многоуровневые отмена и повтор операций, то придется вести список истории выполненных команд. Максимальная длина этого списка и определяет число уровней отмены и повтора. Проход по списку в обратном направлении и отмена результатов всех встретившихся по пути команд отменяет их действие; проход в прямом направлении и выполнение встретившихся команд приводит к повтору действий.

Возможно, команду, допускающую отмену, придется скопировать перед помещением в список истории. Дело в том, что объект команды, использованный для доставки запроса, скажем от пункта меню `MenuItem`, позже мог быть использован для других запросов. Поэтому копирование необходимо, чтобы определить разные вызовы одной и той же команды, если ее состояние при любом вызове может изменяться. Например, команда `DeleteCommand`, которая удаляет выбранные объекты, при каждом вызове должна сохранять разные наборы объектов. Следовательно, объект `DeleteCommand` необходимо скопировать после выполнения, а копию поместить в список истории. Если в результате выполнения состояние команды никогда не изменяется, то копировать не нужно — в список достаточно поместить лишь ссылку на команду. Команды, которые обязательно нужно копировать перед помещением в список истории, ведут себя подобно прототипам (см. описание паттерна прототип (146));

- *предотвращение накопления ошибок в процессе отмены*. При обеспечении надежного, сохраняющего семантику механизма отмены и повтора может возникнуть проблема гистерезиса. При выполнении, отмене и повторе команд иногда накапливаются ошибки, в результате чего состояние приложения оказывается отличным от первоначального. Поэтому порой приходится сохранять в команде больше информации, дабы гарантировать, что объекты будут целиком восстановлены. Чтобы предоставить команде доступ к этой информации, не раскрывая внутреннего устройства объектов, можно воспользоваться паттерном хранитель (330);

- *применение шаблонов в C++*. Для команд, которые: (1) не допускают отмену и (2) не требуют аргументов при вызове, в языке C++ можно воспользоваться шаблонами, чтобы не создавать подкласс класса `Command` для каждой пары «действие — получатель». Как это сделать, мы продемонстрируем в разделе «Пример кода».

Пример кода

Приведенный ниже код на языке C++ дает представление о реализации классов `Command`, обсуждавшихся в разделе «Мотивация». Мы определим классы `OpenCommand`, `PasteCommand` и `MacroCommand`. Начнем с абстрактного класса `Command`:

```
class Command {
public:
    virtual ~Command();
    virtual void Execute() = 0;
protected:
    Command();
};
```

Команда `OpenCommand` открывает документ, имя которому задает пользователь. Конструктору `OpenCommand` передается объект `Application`. Функция `AskUser` запрашивает у пользователя имя открываемого документа:

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);
    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();
    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
```

Команде `PasteCommand` должен передаваться объект `Document`, являющийся получателем. Он передается в параметре конструктора `PasteCommand`:

```
class PasteCommand : public Command {
public:
    PasteCommand(Document*);
```

```
    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc)
{
    _document = doc;
}

void PasteCommand::Execute ()
{
    _document->Paste();
}
```

Для простых команд, не допускающих отмены и не требующих аргументов, можно воспользоваться шаблоном класса для параметризации получателя. Определим для них шаблонный подкласс `SimpleCommand`, который параметризуется типом получателя `Receiver` и хранит связь между объектом-получателем и действием, представленным указателем на функцию класса:

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();
    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }
    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};
```

Конструктор сохраняет информацию о получателе и действии в соответствующих переменных экземпляра. Операция `Execute` просто выполняет действие по отношению к получателю:

```
template <class Receiver>
void SimpleCommand&ltReceiver>::Execute ()
{
    (_receiver->*_action)();
}
```

Чтобы создать команду, которая вызывает операцию `Action` для экземпляра класса `MyClass`, клиент пишет следующий код:

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
```

```
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Имейте в виду, что такое решение годится только для простых команд. Для более сложных команд, которые отслеживают не только получателей, но и аргументы, и, возможно, состояние, необходимое для отмены операции, приходится порождать подклассы от класса `Command`.

Класс `MacroCommand` управляет выполнением последовательности подкоманд и предоставляет операции для добавления и удаления подкоманд. Задавать получателя не требуется, так как в каждой подкоманде уже определен свой получатель:

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();
    virtual void Add(Command*);
    virtual void Remove(Command*);
    virtual void Execute();
private:
    List<Command*>* _cmds;
};
```

Основой класса `MacroCommand` является его функция `Execute`. Она обходит все подкоманды и для каждой вызывает ее операцию `Execute`:

```
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);
    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}
```

Обратите внимание: если бы в классе `MacroCommand` была реализована операция отмены `Unexecute`, то при ее выполнении подкоманды должны были бы отменяться в порядке, обратном порядку их применения в реализации `Execute`.

Наконец, в классе `MacroCommand` должны быть операции для добавления и удаления подкоманд:

```
void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}
```

```
void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}
```

Известные применения

Вероятно, впервые паттерн команда появился в работе Генри Либермана (Henry Lieberman) [Lie85]. В системе MacApp [App89] команды широко применяются для реализации допускающих отмену операций. В ET++ [WGM88], InterViews [LCI+92] и Unidraw [VL90] также имеются классы, описываемые паттерном команда. Так, в библиотеке InterViews определен абстрактный класс `Action`, который определяет функциональность команд. Также определяется шаблон `ActionCallback`, параметризованный методом действия, который автоматически создает экземпляры подклассов команд.

В библиотеке классов THINK [Sym93b] также используются команды для поддержки отмены операций. В THINK команды называются *задачами* (Tasks). Объекты `Task` передаются по цепочке обязанностей (263), пока не будут кем-то обработаны.

Объекты команд в каркасе Unidraw уникальны в том отношении, что могут вести себя подобно сообщениям. В Unidraw команду можно послать другому объекту для интерпретации, результат которой зависит от объекта-получателя. Более того, сам получатель может делегировать интерпретацию следующему объекту, обычно своему родителю. Это напоминает паттерн цепочка обязанностей. Таким образом, в Unidraw получатель вычисляется, а не хранится. Механизм интерпретации в Unidraw использует информацию о типе, доступную во время выполнения.

Джеймс Коплиен описывает, как в языке C++ реализуются *функторы* — объекты, ведущие себя, как функции [Cop92]. Перегрузка оператора вызова `operator()` делает его использование более прозрачным. Смысл паттерна команда в другом — он устанавливает и поддерживает *связь* между получателем и функцией (то есть действием), а не просто функцию.

Родственные паттерны

Паттерн компоновщик (196) можно использовать для реализации макрокоманд.

Паттерн хранитель (330) может сохранять состояние, необходимое команде для отмены ее действия.

Команда, которую нужно копировать перед помещением в список истории, ведет себя, как прототип (146).

ПАТТЕРН INTERPRETER (ИНТЕРПРЕТАТОР)

Название и классификация паттерна

Интерпретатор — паттерн поведения классов.

Назначение

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Мотивация

Если некоторая задача встречается достаточно часто, то имеет смысл представить ее конкретные проявления в виде предложений на простом языке. После этого можно создать интерпретатор, который решает задачу, анализируя предложения этого языка.

Например, поиск строк по образцу — весьма распространенная задача. Регулярные выражения — это стандартный язык для задания образцов поиска. Вместо того чтобы программировать специализированные алгоритмы для сопоставления строк с каждым образцом, алгоритм поиска может интерпретировать регулярное выражение, описывающее множество подходящих строк.

Паттерн интерпретатор определяет грамматику простого языка, представляет предложения на этом языке и интерпретирует их. Для приведенного примера паттерн описывает определение грамматики и интерпретации языка регулярных выражений.

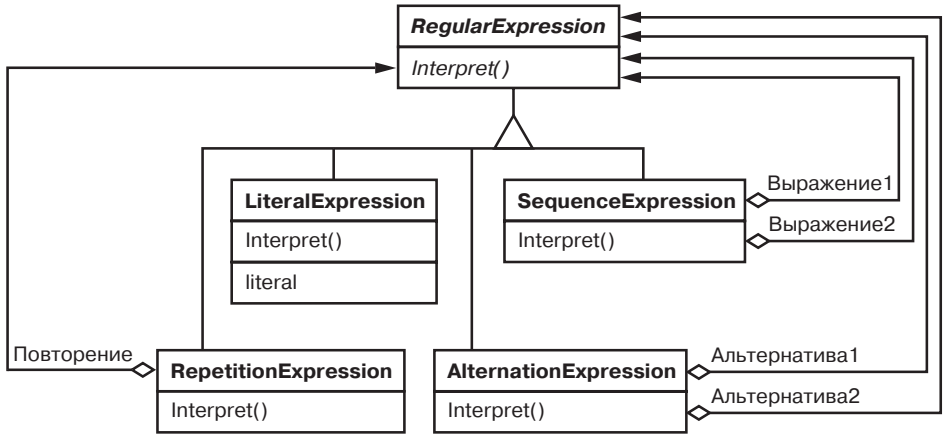
Допустим, они описываются следующей грамматикой:

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

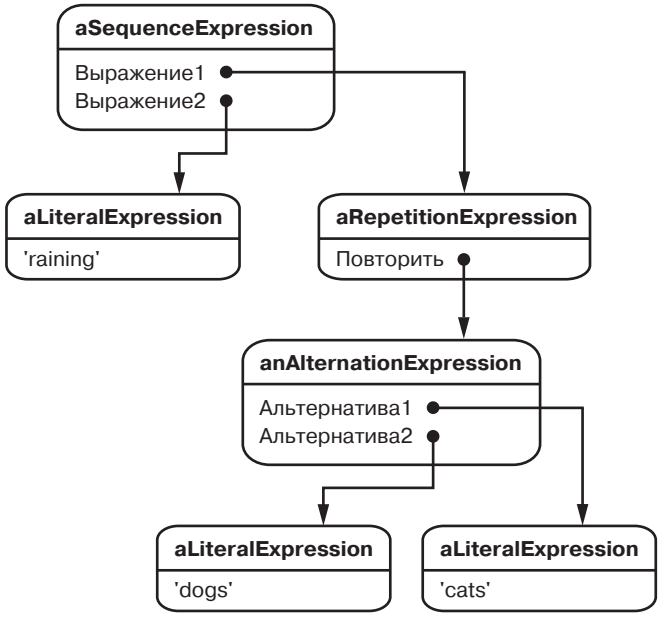
где `expression` — начальный символ, а `literal` — терминальный символ, определяющий простые слова.

Паттерн интерпретатор использует класс для представления каждого правила грамматики. Символы в правой части правила — это переменные экземпляров таких классов. Для представления приведенной выше грамматики требуется пять классов: абстрактный класс `RegularExpression` и четыре его

подкласса `LiteralExpression`, `AlternationExpression`, `SequenceExpression` и `RepetitionExpression`. В последних трех подклассах определены переменные для хранения подвыражений.



Каждое регулярное выражение, описываемое этой грамматикой, представляется в виде абстрактного синтаксического дерева, в узлах которого находятся экземпляры этих классов. Например, дерево



представляет выражение

```
raining & (dogs | cats) *
```

Чтобы создать интерпретатор регулярных выражений, можно определить в каждом подклассе `RegularExpression` операцию `Interpret`, получающую в аргументе контекст, в котором должно интерпретироваться выражение. Контекст состоит из входной строки и информации о текущем состоянии поиска совпадения. В каждом подклассе `RegularExpression` операция `Interpret` ищет совпадение следующей части входной строки с учетом текущего контекста. Например:

- `LiteralExpression` проверяет, соответствует ли входная строка литералу, который хранится в объекте подкласса;
- `AlternationExpression` проверяет, соответствует ли строка одной из альтернатив;
- `RepetitionExpression` проверяет, имеются ли в входной строке повторяющиеся соответствия выражения;

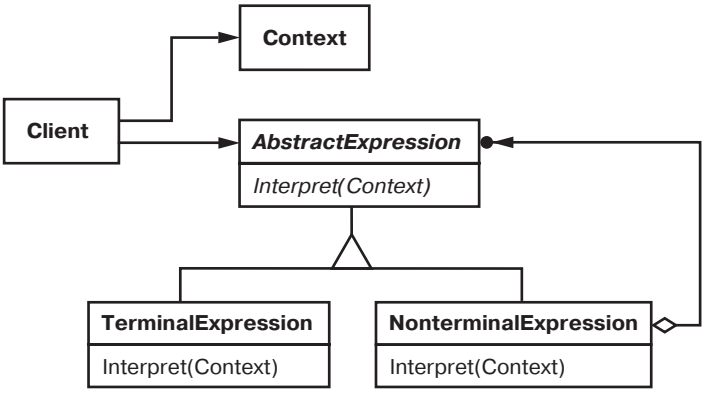
и так далее.

Применимость

Используйте паттерн интерпретатор в ситуациях, когда имеется интерпретируемый язык, конструкции которого можно представить в виде абстрактных синтаксических деревьев. Этот паттерн лучше всего работает в следующих случаях:

- *грамматика проста*. Для сложных грамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять парсеры-генераторы, поскольку они могут интерпретировать выражения без построения абстрактных синтаксических деревьев, что экономит память (а возможно, и время);
- *эффективность не является главным критерием*. Наиболее эффективные интерпретаторы обычно *не* работают непосредственно с деревьями, а сначала транслируют их в другую форму. Так, регулярное выражение часто преобразуется в конечный автомат. Но даже в этом случае сам *транслятор* можно реализовать с помощью паттерна интерпретатор.

Структура



Участники

- **AbstractExpression** (`RegularExpression`) — абстрактное выражение:
  - объявляет абстрактную операцию `Interpret`, общую для всех узлов в абстрактном синтаксическом дереве;
- **TerminalExpression** (`LiteralExpression`) — терминальное выражение:
  - реализует операцию `Interpret` для терминальных символов грамматики;
  - необходим отдельный экземпляр для каждого терминального символа в предложении;
- **NonterminalExpression** (`AlternationExpression`, `RepetitionExpression`, `SequenceExpressions`) — нетерминальное выражение:
  - по одному такому классу требуется для каждого грамматического правила $R :: = R_1R_2...R_n$;
  - хранит переменные экземпляра типа `AbstractExpression` для каждого символа от R_1 до R_n;
  - реализует операцию `Interpret` для нетерминальных символов грамматики. Эта операция рекурсивно вызывает себя же для переменных, представляющих R_1, ... R_n;
- **Context** — контекст:
  - содержит информацию, глобальную по отношению к интерпретатору;

- **Client** — клиент:
  - строит (или получает в готовом виде) абстрактное синтаксическое дерево, представляющее отдельное предложение на языке с данной грамматикой. Дерево собирается из экземпляров классов `NonterminalExpression` и `TerminalExpression`;
  - вызывает операцию `Interpret`.

Отношения

- клиент строит (или получает в готовом виде) конструкцию в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов `NonterminalExpression` и `TerminalExpression`. Затем клиент инициализирует контекст и вызывает операцию `Interpret`;
- в каждом узле вида `NonterminalExpression` через операции `Interpret` определяется операция `Interpret` для каждого подвыражения. Для класса `TerminalExpression` операция `Interpret` определяет базу рекурсии;
- операции `Interpret` в каждом узле используют контекст для сохранения и доступа к состоянию интерпретатора.

Результаты

Основные достоинства и недостатки паттерна интерпретатор:

- *простота изменения и расширения грамматики*. Поскольку для представления грамматических правил в паттерне используются классы, то для изменения или расширения грамматики можно применять наследование. Существующие выражения можно модифицировать постепенно, а новые определять как вариации старых;
- *простая реализация грамматики*. Реализации классов, описывающих узлы абстрактного синтаксического дерева, похожи. Такие классы легко программируются, а зачастую они могут автоматически генерироваться генератором компиляторов или парсером-генератором;
- *сложность сопровождения сложных грамматик*. В паттерне интерпретатор определяется по меньшей мере один класс для каждого правила грамматики (для правил, определенных с помощью формы Бэкуса — Наура — BNF, может понадобиться и более одного класса). Поэтому сопровождение грамматики с большим числом правил иногда оказывается трудной задачей. Для ее решения могут быть применены другие паттерны (см. раздел «Реализация»). Но если грамматика очень сложна,

лучше прибегнуть к другим методам, например воспользоваться генератором компиляторов или парсером-генератором;

- *добавление новых способов интерпретации выражений.* Паттерн интерпретатор позволяет легко изменить способ вычисления выражений. Например, реализовать красивую печать выражения вместо проверки входящих в него типов можно, просто определив новую операцию в классах выражений. Если вам приходится часто создавать новые способы интерпретации выражений, подумайте о применении паттерна посетитель (379). Это поможет избежать изменения классов, описывающих грамматику.

Реализация

У реализаций паттернов интерпретатор и компоновщик (196) много общего. Следующие аспекты относятся только к интерпретатору:

- *создание абстрактного синтаксического дерева.* Паттерн интерпретатор не поясняет, как создавать дерево, то есть разбор выражения не входит в его задачу. Абстрактное дерево разбора можно строить табличноуправляемым или написанным вручную парсером (обычно методом рекурсивного спуска), а также самим клиентом;

- *определение операции Interpret.* Определять операцию `Interpret` в классах выражений необязательно. Если создавать новые интерпретаторы приходится часто, то лучше воспользоваться паттерном посетитель и поместить операцию `Interpret` в отдельный объект-посетитель. Например, для грамматики языка программирования будет нужно определить много операций над абстрактными синтаксическими деревьями: проверку типов, оптимизацию, генерацию кода и т. д. Лучше, конечно, использовать посетителя и не определять эти операции в каждом классе грамматики;

- *разделение терминальных символов с помощью паттерна «приспособленец».* Для грамматик, предложения которых содержат много вхождений одного и того же терминального символа, может оказаться полезным разделение этого символа. Хорошим примером служат грамматики компьютерных программ, поскольку в них каждая переменная встречается в коде многократно. В примере из раздела «Мотивация» терминальный символ *dog* (для моделирования которого используется класс `LiteralExpression`) может встречаться многократно.

  В терминальных узлах обычно не хранится информация о положении в абстрактном синтаксическом дереве. Необходимый для интерпретации

контекст предоставляют им родительские узлы. Налицо различие между разделяемым (внутренним) и передаваемым (внешним) состояниями, так что вполне применим паттерн приспособленец (231).

Например, каждый экземпляр класса `LiteralExpression` для *dog* получает контекст, состоящий из уже просмотренной части строки. И каждый такой экземпляр делает в своей операции `Interpret` одно и то же — проверяет, содержит ли остаток входной строки слово *dog*, — независимо от того, в каком месте дерева этот экземпляр встречается.

Пример кода

Ниже приведены два примера. Первый — законченная программа на Smalltalk для проверки того, существует ли в заданной последовательности совпадение регулярного выражения. Второй — программа на C++ для вычисления булевых выражений.

Программа сопоставления с регулярным выражением проверяет, является ли строка корректным предложением языка, определяемого этим выражением. Регулярное выражение определено следующей грамматикой:

```
expression ::= literal | alternation | sequence | repetition |
'(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression 'repeat'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

Между этой грамматикой и той, что приведена в разделе «Мотивация», есть небольшие отличия. Мы слегка изменили синтаксис регулярных выражений, поскольку в Smalltalk символ * не может быть постфиксной операцией, поэтому вместо него употребляется слово `repeat`. Например, регулярное выражение

```
(('dog ‘ | ‘cat ‘) repeat & ‘weather’)
```

соответствует входной строке `'dog dog cat weather’`.

Для реализации программы сопоставления мы определим пять классов, упомянутых на с. 379. В классе `SequenceExpression` есть переменные экземпляра `expression1` и `expression2` для хранения ссылок на потомков в дереве. Класс `AlternationExpression` хранит альтернативы в переменных экземпляра `alternative1` и `alternative2`, а класс `RepetitionExpression` — повторяемое выражение в переменной экземпляра `repetition`. В классе

`LiteralExpression` есть переменная экземпляра `components` для хранения списка объектов (скорее всего, символов), представляющих литеральную строку, которая должна соответствовать входной строке.

Операция `match:` реализует интерпретатор регулярных выражений. Эта операция реализована в каждом из классов, входящих в абстрактное синтаксическое дерево. Ее аргументом является переменная `inputState`, описывающая текущее состояние процесса сопоставления, то есть уже прочитанную часть входной строки.

Текущее состояние характеризуется множеством входных потоков, представляющим множество тех входных строк, которое регулярное выражение могло бы к настоящему моменту принять. (Это примерно то же, что регистрация всех состояний, в которых побывал бы эквивалентный конечный автомат, распознавший входной поток до данного места.)

Текущее состояние наиболее важно для операции `repeat`. Например, регулярному выражению

```
'a' repeat
```

интерпретатор сопоставил бы строки `"a"`, `"aa"`, `"aaa"` и т. д. А регулярному выражению

```
'a' repeat & 'bc'
```

строки `"abc"`, `"aabc"`, `"aaabc"` и т. д. Но для регулярного выражения

```
'a' repeat & 'abc'
```

сопоставление входной строки `"aabc"` с подвыражением `"'a' repeat"` дало бы два потока, один из которых соответствует одному входному символу, а другой — двум. Из них лишь поток, принявший один символ, может быть сопоставлен с остатком строки `"abc"`.

Теперь рассмотрим определения `match:` для каждого класса, описывающего регулярное выражение. `SequenceExpression` производит сопоставление с каждым подвыражением в определенной последовательности. Обычно потоки ввода не включаются в его состояние `inputState`.

```
match: inputState
    ^ expression2 match: (expression1 match: inputState).
```

`AlternationExpression` возвращает результат, объединяющий состояния всех альтернатив. Определение `match:` для этого случая выглядит так:

```
match: inputState
    | finalState |
    finalState := alternative1 match: inputState.
    finalState addAll: (alternative2 match: inputState).
    ^ finalState
```

Операция `match:` для `RepetitionExpression` пытается найти максимальное количество состояний, допускающих сопоставление:

```
match: inputState
    | aState finalState |
    aState := inputState.
    finalState := inputState copy.
    [aState isEmpty]
        whileFalse:
            [aState := repetition match: aState.
            finalState addAll: aState].
        ^ finalState
```

На выходе этой операции мы обычно получаем больше состояний, чем на входе, поскольку `RepetitionExpression` может быть сопоставлено с одним, двумя или более вхождениями повторяющегося выражения во входную строку. В выходном состоянии представлены все возможные варианты, а решение о том, какое из состояний правильно, принимается последующими элементами регулярного выражения.

Наконец, операция `match:` для `LiteralExpression` сравнивает свои компоненты с каждым возможным входным потоком и оставляет только те из них, для которых попытка завершилась удачно:

```
match: inputState
    | finalState tStream |
    finalState := Set new.
    inputState
        do:
            [:stream | tStream := stream copy.
                (tStream nextAvailable:
                    components size
                ) = components
                    ifTrue: [finalState add: tStream]
            ].
        ^ finalState
```

Сообщение `nextAvailable:` выполняет смещение вперед по входному потоку. Это единственная операция `match:`, которая сдвигается по потоку. Обратите внимание: возвращаемое состояние содержит его копию, поэтому можно быть

уверенным, что сопоставление с литералом никогда не изменяет входной поток. Это существенно, поскольку все альтернативы в `AlternationExpression` должны «видеть» идентичные копии входного потока.

Итак, классы, составляющие абстрактное синтаксическое дерево, определены; теперь опишем процесс его построения. Вместо того чтобы создавать парсер регулярных выражений, мы определим некоторые операции в классах `RegularExpression`, так что вычисление выражения языка Smalltalk приведет к созданию абстрактного синтаксического дерева для соответствующего регулярного выражения. Тем самым мы будем использовать встроенный компилятор Smalltalk, как если бы это был парсер для регулярных выражений.

Для построения дерева нам понадобятся операции `"|"`, `"repeat"` и `"&"` над регулярными выражениями. Определим эти операции в классе `RegularExpression`:

```
& aNode
    ^ SequenceExpression new
        expression1: self expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self
    | aNode
        ^ AlternationExpression new
        alternative1: self alternative2: aNode asRExp
asRExp
    ^ self
```

Операция `asRExp` преобразует литералы в `RegularExpression`. Следующие операции определены в классе `String`:

```
& aNode
    ^ SequenceExpression new
        expression1: self asRExp expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
        alternative1: self asRExp alternative2: aNode asRExp

asRExp
    ^ LiteralExpression new components: self
```

Если бы эти операции были определены выше в иерархии классов (`SequenceableCollection` в Smalltalk-80, `IndexedCollection` в Smalltalk/V),

то они появились бы и в таких классах, как Array и OrderedCollection. Это позволило бы сопоставлять регулярные выражения с последовательностями объектов любого вида.

Второй пример — это система для манипулирования и вычисления булевых выражений, реализованная на C++. Терминальными символами в этом языке являются булевы переменные, то есть константы true и false. Нетерминальные символы представляют выражения, содержащие операторы and, or и not. Определение грамматики выглядит так[1]:

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
    '(' BooleanExp ')'
AndExp ::= BooleanExp  'and' BooleanExp
OrExp ::= BooleanExp  'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' |  'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

Определим две операции над булевыми выражениями. Первая — Evaluate — вычисляет выражение в контексте, где каждой переменной присваивается истинное или ложное значение. Вторая — Replace — порождает новое булево выражение, заменяя выражением некоторую переменную. Эта операция демонстрирует, что паттерн интерпретатор можно использовать не только для вычисления выражений; в данном случае он манипулирует самим выражением.

Здесь подробно описываются только классы BooleanExp, VariableExp и AndExp. Классы OrExp и NotExp аналогичны классу AndExp. Класс Constant представляет булевы константы.

В классе BooleanExp определен интерфейс всех классов, которые описывают булевы выражения:

```
class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();
    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};
```

[1] Для простоты мы игнорируем приоритеты операторов и предполагаем, что этой проблемой должен заниматься объект, строящий дерево разбора.

Класс `Context` определяет соответствие между переменными и булевыми значениями, которые в C++ представляются константами `true` и `false`. Возьмем следующий интерфейс:

```
class Context {
public:
    bool Lookup(const char*) const;
    void Assign(VariableExp*, bool);
};
```

Класс `VariableExp` представляет именованную переменную:

```
class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();
    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};
```

Конструктор класса получает в аргументе имя переменной:

```
VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}
```

Вычисление переменной возвращает ее значение в текущем контексте:

```
bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}
```

Копирование переменной возвращает новый объект класса `VariableExp`:

```
BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}
```

Чтобы заменить переменную выражением, мы сначала проверяем, что имя переменной совпадает с именем, переданным в аргументе:

```
BooleanExp* VariableExp::Replace ( const char* name, BooleanExp& exp ) {
    if (strcmp(name, _name) == 0) {
        return exp.Copy();
```

```
    } else {
        return new VariableExp(_name);
    }
}
```

Класс `AndExp` представляет выражение, получающееся в результате применения операции логического И к двум булевым выражениям:

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~ AndExp();
    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}
```

При решении `AndExp` вычисляются его операнды и возвращается результат применения к ним операции логического И:

```
bool AndExp::Evaluate (Context& aContext) {
    return _operand1->Evaluate(aContext) &&
           _operand2->Evaluate(aContext);
}
```

В классе `AndExp` операции `Copy` и `Replace` реализуются с помощью рекурсивных обращений к операндам:

```
BooleanExp* AndExp::Copy () const {
    return new AndExp(_operand1->Copy(), _operand2->Copy());
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
        new AndExp(
            _operand1->Replace(name, exp),
            _operand2->Replace(name, exp)
        );
}
```

Определим теперь булево выражение

```
(true and x) or (y and (not x))
```

и вычислим его для некоторых конкретных значений булевых переменных x и y:

```
BooleanExp* expression;
Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);
```

С такими значениями x и y выражение равно `true`. Чтобы вычислить его при других значениях переменных, достаточно просто изменить контекст.

Наконец, можно заменить переменную y новым выражением и повторить вычисление:

```
VariableExp* z = new VariableExp("Z");
NotExp not_z(z);
BooleanExp* replacement = expression->Replace("Y", not_z);
context.Assign(z, true);
result = replacement->Evaluate(context);
```

Этот пример иллюстрирует важную особенность паттерна интерпретатор: многие разновидности операций могут «интерпретировать» последовательности. Из трех операций, определенных в классе `BooleanExp`, `Evaluate` наиболее близка к нашему интуитивному представлению о том, что интерпретатор должен интерпретировать программу или выражение и возвращать простой результат.

Но и операцию `Replace` можно считать интерпретатором. Его контекстом является имя заменяемой переменной и подставляемое вместо него выражение, а результатом служит новое выражение. Даже операция `Copy` может рассматриваться как интерпретатор с пустым контекстом. Трактовка операций `Replace` и `Copy` как интерпретаторов может показаться странной, поскольку

это всего лишь базовые операции над деревом. Примеры в описании паттерна посетитель (379) демонстрируют, что все три операции разрешается вынести в отдельный объект-посетитель «интерпретатор», тогда аналогия станет более очевидной.

Паттерн интерпретатор — нечто большее, чем распределение некоторой операции по иерархии классов, составленной с помощью паттерна компоновщик (196). Мы рассматриваем операцию `Evaluate` как интерпретатор, поскольку иерархию классов `BooleanExp` мыслим себе как представление некоторого языка. Если бы у нас была аналогичная иерархия для представления агрегатов автомобиля, то вряд ли мы стали бы рассматривать такие операции, как `Weight` (вес) и `Copy` (копирование), как интерпретаторы, несмотря на то что они распределены по всей иерархии классов, — просто мы не воспринимаем части автомобиля как язык. Тут все дело в точке зрения: опубликуй мы грамматику автомобильных частей, то операции над ними можно было трактовать как способы интерпретации соответствующего языка.

Известные применения

Паттерн интерпретатор широко используется в компиляторах, реализованных с помощью объектно-ориентированных языков, например в компиляторах Smalltalk. В языке SPECTalk этот паттерн применяется для интерпретации форматов входных файлов [Sza92]. В библиотеке QOCA он применяется для вычисления ограничений [HHMV92].

Если рассматривать данный паттерн в самом общем виде (то есть как операцию, распределенную по иерархии классов, основанной на паттерне компоновщик), то почти любое применение компоновщика содержит и интерпретатор. Но паттерн интерпретатор лучше применять в тех случаях, когда иерархию классов можно рассматривать как описание языка.

Родственные паттерны

Компоновщик (196): абстрактное синтаксическое дерево — пример применения паттерна компоновщик.

Приспособленец (231) показывает варианты разделения терминальных символов в абстрактном синтаксическом дереве.

Итератор (302): интерпретатор может пользоваться итератором для обхода структуры.

Посетитель (379) может использоваться для инкапсуляции в одном классе поведения каждого узла абстрактного синтаксического дерева.

ПАТТЕРН ITERATOR (ИТЕРАТОР)

Название и классификация паттерна

Итератор — паттерн поведения объектов.

Назначение

Предоставляет способ последовательного обращения ко всем элементам составного объекта без раскрытия его внутреннего представления.

Другие названия

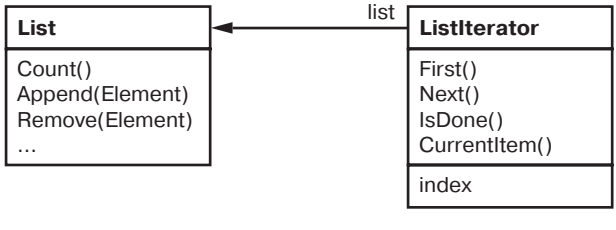
`Cursor` (курсор).

Мотивация

Составной объект, скажем, список, должен предоставлять способ доступа к своим элементам, не раскрывая их внутреннюю структуру. Более того, иногда требуется обходить список по-разному в зависимости от решаемой задачи. Но вряд ли вы захотите засорять интерфейс класса `List` операциями для различных вариантов обхода, даже если все их можно предусмотреть заранее. Кроме того, иногда бывает нужно, чтобы в один и тот же момент действовало несколько активных обходов списка.

Все это позволяет сделать паттерн итератор. Основная его идея в том, чтобы за обращения к элементам и способ обхода отвечал не сам список, а отдельный объект-итератор. В классе `Iterator` определен интерфейс для доступа к элементам списка. Объект этого класса отслеживает текущий элемент, то есть он располагает информацией, какие элементы уже посещались.

Например, для класса `List` мог бы существовать класс `ListIterator`; отношения между этими классами могли бы выглядеть так:



Прежде чем создавать экземпляр класса `ListIterator`, необходимо иметь список для обхода. С объектом `ListIterator` вы можете последовательно

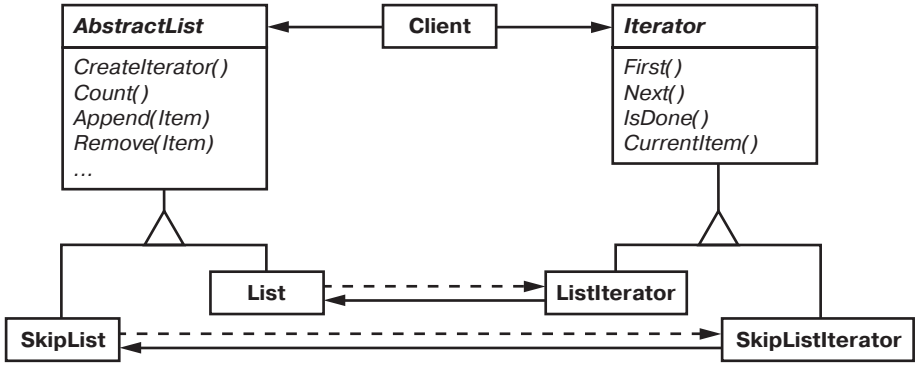
посетить все элементы списка. Операция `CurrentItem` возвращает текущий элемент списка, `First` инициализирует текущий элемент первым элементом списка, `Next` делает текущим следующий элемент, а `IsDone` проверяет, не вышел ли обход за последний элемент, если вышел — то обход завершается.

Отделение механизма обхода от объекта `List` позволяет определять итераторы, реализующие различные стратегии обхода, не перечисляя их в интерфейсе класса `List`. Например, итератор `FilteringListIterator` мог бы предоставлять доступ только к тем элементам, которые удовлетворяют критериям фильтрации.

Следует заметить, что между итератором и списком существует тесная связь. Клиент должен знать, что он обходит именно *список*, а не какую-то другую агрегированную структуру. По этой причине клиент привязан к конкретному способу агрегирования. Было бы лучше, если бы класс агрегата можно было изменять без изменения кода клиента. Для этого можно обобщить концепцию итератора и рассмотреть *полиморфную итерацию*.

Допустим, у вас есть еще класс `SkipList`, реализующий *список с пропусками* (skiplist) [Pug90] — вероятностную структуру данных, которая по своим характеристикам напоминает сбалансированное дерево. Требуется иметь возможность писать код, способный работать с объектами как класса `List`, так и класса `SkipList`.

Определим класс `AbstractList`, в котором объявлен общий интерфейс для манипулирования списками. Еще нам понадобится абстрактный класс `Iterator`, определяющий общий интерфейс итерации. Затем мы смогли бы определить конкретные подклассы класса `Iterator` для различных реализаций списка. В результате механизм итерации перестает зависеть от конкретных агрегированных классов.



Остается понять, как создается итератор. Поскольку мы хотим написать код, не зависящий от конкретных подклассов `List`, то нельзя просто создать экземпляр конкретного класса. Вместо этого ответственность за создание подходящих объектов-списков будет возложена на сами объекты-списки; вот почему потребуется операция `CreateIterator`, посредством которой клиенты смогут запрашивать объект-итератор.

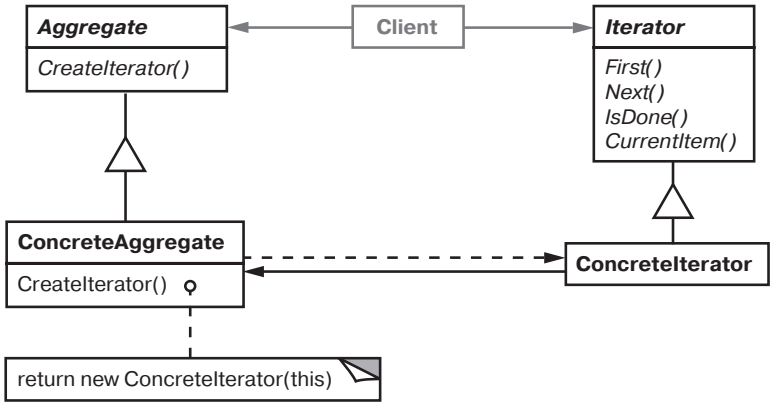
`CreateIterator` — это пример использования паттерна фабричный метод (135). В данном случае он служит для того, чтобы клиент мог запросить у объекта-списка подходящий итератор. Применение фабричного метода приводит к появлению двух иерархий классов — одной для списков, другой для итераторов. Фабричный метод `CreateIterator` связывает эти две иерархии.

Применимость

Основные условия для применения паттерна итератор:

- обращение к содержимому агрегированных объектов без раскрытия их внутреннего представления;
- поддержка нескольких активных обходов одного и того же агрегированного объекта;
- предоставление единообразного интерфейса для обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).

Структура



Участники

- **Iterator** — итератор:
  - определяет интерфейс для доступа и обхода элементов;

- **ConcreteIterator** — конкретный итератор:
  - реализует интерфейс класса `Iterator`;
  - следит за текущей позицией при обходе агрегата;
- **Aggregate** — агрегат:
  - определяет интерфейс для создания объекта-итератора;
- **ConcreteAggregate** — конкретный агрегат:
  - реализует интерфейс создания итератора и возвращает экземпляр подходящего класса `ConcreteIterator`.

Отношения

`ConcreteIterator` отслеживает текущий объект в агрегате и может вычислить идущий за ним.

Результаты

Основные достоинства и недостатки паттерна итератор:

- *поддержка разных способов обхода агрегата*. Сложные агрегаты можно обходить по-разному. Например, для генерации кода и семантических проверок нужно обходить деревья синтаксического разбора. Генератор кода может обходить дерево во внутреннем или прямом порядке. Итераторы упрощают изменение алгоритма обхода — достаточно просто заменить один экземпляр итератора другим. Для поддержки новых видов обхода можно определить и подклассы класса `Iterator`;
- *упрощение интерфейса класса Aggregate*. Наличие интерфейса для обхода в классе Iterator делает излишним дублирование этого интерфейса в классе `Aggregate`. Тем самым интерфейс агрегата упрощается;
- *возможность наличия нескольких активных обходов для данного агрегата*. Итератор следит за инкапсулированным в нем самом состоянием обхода, поэтому одновременно могут существовать несколько обходов агрегата.

Реализация

Существует множество вариантов и альтернативных способов реализации итератора, ниже перечислены наиболее употребительные. Выбор часто зависит от управляющих структур, поддерживаемых языком программирования. Некоторые языки (например, CLU [LG86]) даже поддерживают данный паттерн напрямую.

- *Какой участник управляет итерацией?* Важнейший вопрос состоит в том, что управляет итерацией: сам итератор или клиент, который им пользуется. Если итерацией управляет клиент, то итератор называется *внешним*, в противном случае — *внутренним*[1]. Клиенты, применяющие внешний итератор, должны явно запросить у итератора следующий элемент, чтобы двигаться дальше по агрегату. С другой стороны, в случае внутреннего итератора клиент передает итератору некоторую операцию, а итератор уже сам применяет эту операцию к каждому посещенному во время обхода элементу агрегата.

  Внешние итераторы обладают большей гибкостью, чем внутренние. Например, сравнить две коллекции на равенство с помощью внешнего итератора очень легко, а с помощью внутреннего — практически невозможно. Слабые стороны внутренних итераторов наиболее отчетливо проявляются в таких языках, как C++, где нет анонимных функций, замыканий (closure) и продолжений (continuation), как в Smalltalk или CLOS. С другой стороны, внутренние итераторы проще в использовании, поскольку они вместо вас определяют логику обхода;

- *что определяет алгоритм обхода*? Алгоритм обхода можно определить не только в итераторе. Его может определить сам агрегат и использовать итератор только для хранения состояния итерации. Такого рода итератор называется *курсором*, поскольку он всего лишь указывает на текущую позицию в агрегате. Клиент вызывает операцию `Next` агрегата, передавая ей курсор в качестве аргумента. Операция же `Next` изменяет состояние курсора[2].

  Если за алгоритм обхода отвечает итератор, то для одного и того же агрегата можно использовать разные алгоритмы итерации, и, кроме того, проще применить один алгоритм к разным агрегатам. С другой стороны, алгоритму обхода может понадобиться доступ к закрытым переменным агрегата. Если это так, то перенос алгоритма в итератор нарушает инкапсуляцию агрегата;

- *насколько итератор устойчив*? Модификация агрегата в то время, как совершается его обход, может оказаться опасной. Если при этом добавляются или удаляются элементы, то не исключено, что некоторый

---

[1] Грейди Буч (Grady Booch) называет внешние и внутренние итераторы соответственно активными и пассивными [Boo94]. Термины «активный» и «пассивный» относятся к роли клиента, а не к действиям, выполняемым итератором.

[2] Курсоры — это простой пример применения паттерна хранитель; их реализации имеют много общего.

элемент будет посещен дважды или вообще ни разу. Простое решение — скопировать агрегат и обходить копию, но обычно это слишком дорого.

Устойчивый итератор (robust) гарантирует, что ни вставки, ни удаления не помешают обходу, причем достигается это без копирования агрегата. Есть много способов реализации устойчивых итераторов. В большинстве из них итератор регистрируется в агрегате. При вставке или удалении агрегат либо корректирует внутреннее состояние всех созданных им итераторов, либо организует внутреннюю информацию так, чтобы обход выполнялся правильно.

В работе Томаса Кофлера (Thomas Kofler) [Kof93] приводится подробное обсуждение реализации итераторов в каркасе ET++. Роберт Мюррей (Robert Murray) [Mur93] описывает реализацию устойчивых итераторов для класса `List` из библиотеки USL Standard Components;

- *дополнительные операции итератора*. Минимальный интерфейс класса `Iterator` состоит из операций `First`, `Next`, `IsDone` и `CurrentItem`[1]. Впрочем, некоторые дополнительные операции также могут оказаться полезными. Например, упорядоченные агрегаты могут предоставлять операцию `Previous`, переводящую итератор к предыдущему элементу. Для отсортированных или индексированных коллекций интерес представляет операция `SkipTo`, которая позиционирует итератор на объект, удовлетворяющий некоторому критерию;
- *использование полиморфных итераторов в C++*. Использование полиморфных итераторов сопряжено с определенными затратами. Объект-итератор должен создаваться динамически фабричным методом, поэтому использовать их стоит только тогда, когда есть необходимость в полиморфизме. В противном случае применяйте конкретные итераторы, которые вполне можно создавать в стеке.

  У полиморфных итераторов есть еще один недостаток: за их удаление отвечает клиент. Здесь открывается большой простор для ошибок, так как очень легко забыть об освобождении созданного в куче объекта-итератора после завершения работы с ним. Особенно велика вероятность этого, если у операции есть несколько точек выхода. А в случае выдачи исключения память, занимаемая объектом-итератором, вообще никогда не будет освобождена.

---

[1] Этот интерфейс можно дополнительно сократить, объединив операции Next, IsDone и CurrentItem в одну, которая будет переходить к следующему объекту и возвращать его. Если обход завершен, то операция вернет специальное значение (например, 0), обозначающее конец итерации.

Ситуацию помогает исправить паттерн заместитель (246). Вместо настоящего итератора используется его заместитель, память для которого выделяется в стеке. Заместитель уничтожает итератор в своем деструкторе, поэтому как только заместитель выходит из области видимости, вместе с ним уничтожается и настоящий итератор. Заместитель гарантирует выполнение надлежащей очистки даже при возникновении исключений. Это пример применения хорошо известного в C++ принципа «инициализации при создании ресурса» [ES90]. В разделе «Пример кода» он проиллюстрирован подробнее;

- *возможность привилегированного доступа к итераторам.* Итератор можно рассматривать как расширение создавшего его агрегата. Итератор и агрегат тесно связаны. В C++ такую связь можно выразить, назначив итератор *другом* своего агрегата. Тогда не нужно определять в агрегате операции, единственная цель которых — позволить итераторам эффективно выполнить обход.

  Однако наличие такого привилегированного доступа может затруднить определение новых способов обхода, так как потребуется изменить интерфейс агрегата, добавив в него нового друга. Для решения этой проблемы класс `Iterator` может включать защищенные операции для доступа к важным, но не являющимся открытыми членам агрегата. Подклассы класса Iterator (и *только* его подклассы) могут воспользоваться этими защищенными операциями для получения привилегированного доступа к агрегату;

- *итераторы для составных объектов.* Реализовать внешние агрегаты для рекурсивно агрегированных структур (таких, например, которые возникают в результате применения паттерна компоновщик (196)) может оказаться затруднительно, поскольку описание положения в структуре иногда охватывает несколько уровней вложенности. Поэтому, чтобы отследить позицию текущего объекта, внешний итератор должен хранить путь через составной объект `Composite`. Иногда проще воспользоваться внутренним итератором. Он может запомнить текущую позицию, рекурсивно вызывая себя самого, так что путь будет неявно храниться в стеке вызовов.

  Если узлы составного объекта `Composite` имеют интерфейс для перемещения от узла к его братьям, родителям и потомкам, то итератор курсорного типа может оказаться более достойной альтернативой. Курсору нужно следить только за текущим узлом, а для обхода составного объекта он может положиться на интерфейс этого узла.

Составные объекты часто нужно обходить несколькими способами. Самые распространенные — это обход в прямом, обратном и внутреннем порядке, а также обход в ширину. Каждый вид обхода может поддерживаться отдельным итератором;

- *пустые итераторы*. Пустой итератор `NullIterator` представляет собой вырожденный итератор, полезный при обработке граничных условий. По определению, `NullIterator` *всегда* считает, что обход завершен, то есть его операция `IsDone` неизменно возвращает `true`.

  Пустой итератор может упростить обход древовидных структур (например, объектов `Composite`). В каждой точке обхода у текущего элемента запрашивается итератор для его потомков. Элементы-агрегаты, как обычно, возвращают конкретный итератор, но листовые элементы возвращают экземпляр `NullIterator`. Это позволяет реализовать обход всей структуры единообразно.

Пример кода

Рассмотрим простой класс списка `List`, входящего в нашу базовую библиотеку (см. приложение B), и две реализации класса `Iterator`: одну для обхода списка от начала к концу, а другую — от конца к началу (в базовой библиотеке поддерживается только первый способ). Затем будет показано, как пользоваться этими итераторами и как избежать зависимости от конкретной реализации. После этого изменим дизайн, дабы гарантировать корректное удаление итераторов. А в последнем примере мы проиллюстрируем внутренний итератор и сравним его с внешним.

- *Интерфейсы классов List и Iterator*. Сначала обсудим ту часть интерфейса класса `List`, которая имеет отношение к реализации итераторов. Полный интерфейс представлен в приложении B:

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);
    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

  Открытый интерфейс класса `List` предоставляет эффективный способ поддержки итераций. Его достаточно для реализации обоих способов обхода. Поэтому нет необходимости предоставлять итераторам привилегированный доступ к внутренней структуре данных. Иными словами, классы

итераторов не являются друзьями класса `List`. Определим абстрактный класс `Iterator`, в котором будет объявлен интерфейс итератора:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

- *реализации подклассов класса Iterator.* Класс `ListIterator` является подклассом `Iterator`:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

Реализация класса `ListIterator` достаточно прямолинейна. В нем хранится экземпляр `List` и индекс текущей позиции в списке `_current`:

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

Операция `First` позиционирует итератор на первый элемент списка:

```
template <class Item>
void ListIterator<Item>::First ()
{
    _current = 0;
}
```

Операция `Next` делает текущим следующий элемент:

```
template <class Item>
void ListIterator<Item>::Next ()  {
    _current++;
}
```

Операция `IsDone` проверяет, относится ли индекс к элементу, находящемуся в списке:

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

Наконец, операция `CurrentItem` возвращает элемент, соответствующий текущему индексу. Если итерация уже завершилась, то выдается исключение `IteratorOutOfBounds`:

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

Реализация обратного итератора `ReverseListIterator` аналогична рассмотренной, только его операция `First` позиционирует `_current` на конец списка, а операция `Next` делает текущим предыдущий элемент;

- *использование итераторов*. Предположим, имеется список объектов `Employee` (служащий) и мы хотели бы напечатать информацию обо всех содержащихся в нем служащих. Класс `Employee` поддерживает печать с помощью операции `Print`. Для печати списка определим операцию `PrintEmployees`, получающую в аргументе итератор. Она пользуется этим итератором для обхода и вывода содержимого списка:

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

Поскольку у нас есть итераторы для обхода списка от начала к концу и от конца к началу, та же операция может повторно использоваться для вывода списка в обоих направлениях:

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

- *предотвращение привязки к конкретной реализации списка*. Рассмотрим, как повлияла бы на код итератора реализация класса `List` в виде списка с пропусками. Подкласс `SkipList` класса `List` должен предоставить итератор `SkipListIterator`, реализующий интерфейс класса `Iterator`. Для эффективной реализации итерации у `SkipListIterator` должен быть не только индекс. Но поскольку `SkipListIterator` согласуется с интерфейсом класса `Iterator`, то операцию `PrintEmployees` можно использовать и тогда, когда служащие хранятся в списке типа `SkipList`:

```
SkipList<Employee*>* employees;
// ...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

Такое решение работает, но вообще лучше не привязываться к конкретной реализации списка (например, `SkipList`). Можно рассмотреть абстрактный класс `AbstractList` ради стандартизации интерфейса списка для различных реализаций. Тогда и `List`, и `SkipList` окажутся подклассами `AbstractList`.

Для поддержки полиморфной итерации класс `AbstractList` определяет фабричный метод `CreateIterator`, замещаемый в подклассах, которые возвращают подходящий для себя итератор:

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

Альтернативный вариант — определение общего класса-примеси `Traversable`, в котором определен интерфейс для создания итератора.

Для поддержки полиморфных итераций агрегированные классы могут являться потомками `Traversable`.

Класс `List` замещает `CreateIterator` так, чтобы операция возвращала объект `ListIterator`:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Теперь можно написать код для вывода списка, который не будет зависеть от конкретного представления списка:

```
// Известно только то, что объект относится к классу AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

- *гарантированное удаление итераторов*. Заметим, что `CreateIterator` возвращает только что созданный в динамической памяти объект-итератор. Ответственность за его удаление лежит на нас, и если вы забудете это сделать, то возникнет утечка памяти. Чтобы упростить задачу клиентам, введем класс `IteratorPtr`, который замещает итератор. Он уничтожит объект `Iterator` при выходе из области видимости.

  Объект класса `IteratorPtr` всегда создается в стеке[1]. C++ автоматически вызовет его деструктор, который уничтожит реальный итератор. В классе `IteratorPtr` операторы `operator->` и `operator*` перегружены так, что объект этого класса может рассматриваться как указатель на итератор. Функции класса `IteratorPtr` реализуются как встроенные, поэтому они не создают лишних затрат ресурсов:

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
```

[1] Это можно проверять на этапе компиляции, если объявить операторы `new` и `delete` закрытыми. Реализовывать их при этом не надо.

```
    Iterator<Item>& operator*() { return *_i; }
private:
    // Запретить копирование и присваивание, чтобы
    // избежать многократных удалений _i:
    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};
```

`IteratorPtr` позволяет упростить код печати:

```
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

- *внутренний ListIterator*. В последнем примере рассмотрим, как можно было бы реализовать внутренний (или пассивный) класс `ListIterator`. В этом случае итератор сам управляет итерацией и применяет к каждому элементу некоторую операцию.

  Проблема в том, как параметризовать итератор той операцией, которую мы хотим применить к каждому элементу. C++ не поддерживает ни анонимных функций, ни замыканий, которые предусмотрены для этой цели в других языках. Есть по крайней мере два варианта: (1) передать указатель на функцию (глобальную или статическую) и (2) породить подклассы. В первом случае итератор вызывает переданную ему операцию в каждой точке обхода. Во втором случае итератор вызывает операцию, которая замещена в подклассе и обеспечивает нужное поведение.

  Ни один из вариантов не идеален. Часто во время обхода нужно накапливать некоторую информацию, а функции для этого плохо подходят — для запоминания состояния пришлось бы использовать статические переменные. Подкласс класса `Iterator` предоставляет удобное место для хранения накопленного состояния — переменную экземпляра. Однако создавать подкласс для каждого вида обхода слишком трудоемко.

  Ниже приведен черновик реализации второго варианта с использованием подклассов. Назовем внутренний итератор `ListTraverser`:

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
```

```
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

`ListTraverser` получает экземпляр `List` в параметре. Во внутренней реализации он использует внешний итератор `ListIterator` для выполнения обхода. Операция `Traverse` начинает обход и вызывает для каждого элемента операцию `ProcessItem`. Внутренний итератор может закончить обход, вернув `false` из `ProcessItem`. `Traverse` сообщает о преждевременном завершении обхода:

```
template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```

Воспользуемся итератором `ListTraverser` для вывода первых десяти элементов списка. С этой целью надо породить подкласс от `ListTraverser` и определить в нем операцию `ProcessItem`. Для подсчета выведенных элементов используется переменная экземпляра `_count`:

```
class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }
```

```
protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}
```

Вот как `PrintNEmployees` выводит первые 10 элементов:

```
List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Обратите внимание, что в коде клиента нет цикла итерации. Всю логику обхода можно использовать повторно. В этом и состоит основное преимущество внутреннего итератора. Правда, работы немного больше, чем для внешнего итератора, так как нужно определять новый класс. Сравните с программой, где применяется внешний итератор:

```
ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
    count++;
    i.CurrentItem()->Print();

    if (count >= 10) {
        break;
    }
}
```

Внутренние итераторы могут инкапсулировать разные виды итераций. Например, `FilteringListTraverser` инкапсулирует итерацию, при которой обрабатываются лишь элементы, удовлетворяющие определенному условию:

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
```

```
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

Интерфейс такой же, как у `ListTraverser`, если не считать новой функции, которая и реализует проверку условия. В подклассах `TestItem` замещается для задания конкретного условия.

Операция `Traverse` выясняет, нужно ли продолжать обход по результатам проверки:

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());

            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```

В качестве варианта можно было определить функцию `Traverse` так, чтобы она сообщала хотя бы об одном встретившемся элементе, который удовлетворяет условию[1].

Известные применения

Итераторы широко распространены в объектно-ориентированных системах. В том или ином виде они поддерживаются в большинстве библиотек коллекций классов.

[1] Операция `Traverse` в этих примерах — это ничто иное, как шаблонный метод с примитивными операциями `TestItem` и `ProcessItem`..

Вот пример из библиотеки компонентов Грейди Буча [Boo94], популярной библиотеки, поддерживающей классы коллекций. В ней имеется реализация очереди фиксированной (ограниченной) и динамически растущей длины (неограниченной). Интерфейс очереди определен в абстрактном классе `Queue`. Для поддержки полиморфной итерации по очередям с разной реализацией итератор написан с использованием интерфейса абстрактного класса `Queue`. Преимущество такого подхода очевидно — отпадает необходимость в фабричном методе, который запрашивал бы у очереди соответствующий ей итератор. Но чтобы итератор можно было реализовать эффективно, интерфейс абстрактного класса `Queue` должен быть достаточно мощным.

В языке Smalltalk необязательно определять итераторы так явно. В стандартных классах коллекций (`Bag`, `Set`, `Dictionary`, `OrderedCollection`, `String` и т. д.) определен метод `do:`, выполняющий функции внутреннего итератора, который получает блок (то есть замыкание) в аргументе. Каждый элемент коллекции привязывается к локальной переменной в блоке, после чего блок выполняется. Smalltalk также включает набор классов `Stream`, которые поддерживают интерфейс, сходный с интерфейсом итераторов. `ReadStream` — это фактически класс `Iterator` и внешний итератор для всех последовательных коллекций. У непоследовательных коллекций (таких как `Set` и `Dictionary`) нет стандартных итераторов.

Полиморфные итераторы и выполняющие очистку заместители находятся в контейнерных классах ET++ [WGM88]. Курсороподобные итераторы используются в классах каркаса графических редакторов Unidraw [VL90].

В системе ObjectWindows 2.0 [Bor94] имеется иерархия классов итераторов для контейнеров. Контейнеры разных типов можно обходить одним и тем же способом. Синтаксис итераторов в ObjectWindows основан на перегрузке постфиксного оператора инкремента `++` для перехода к следующему элементу.

Родственные паттерны

Компоновщик (196): итераторы довольно часто применяются для обхода рекурсивных структур, создаваемых компоновщиком.

Фабричный метод (135): полиморфные итераторы поручают фабричным методам создавать экземпляры подходящих подклассов класса `Iterator`.

Итератор (302) может использовать хранитель для сохранения состояния итерации и при этом содержит его внутри себя.

ПАТТЕРН MEDIATOR (ПОСРЕДНИК)

Название и классификация паттерна

Посредник — паттерн поведения объектов.

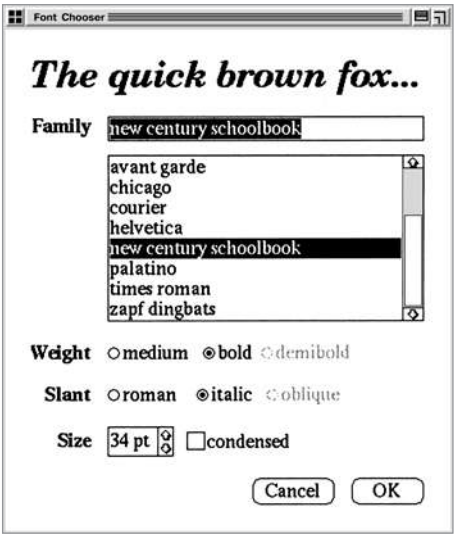
Назначение

Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

Мотивация

Объектно-ориентированное проектирование способствует распределению некоторого поведения между объектами. Но при этом в получившейся структуре объектов может возникнуть много связей или (в худшем случае) каждый объект должен располагать информацией обо всех остальных.

Хотя разбиение системы на множество объектов в общем случае повышает степень повторного использования, размножение взаимосвязей приводит к обратному эффекту. Если взаимосвязей слишком много, тогда система подобна монолиту и маловероятно, что объект сможет работать без поддержки других объектов. Более того, существенно изменить поведение системы практически невозможно, поскольку оно распределено между многими объектами. В результате для настройки поведения системы вам придется определять множество подклассов.



Для примера рассмотрим реализацию диалоговых окон в графическом интерфейсе пользователя. Здесь располагается ряд виджетов: кнопки, меню, поля ввода и т. д., как показано на рисунке.

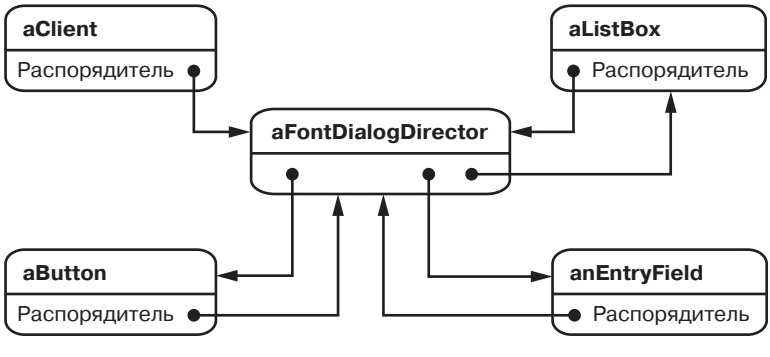
Часто между виджетами в диалоговом окне существуют зависимости. Например, если одно из полей ввода пустое, то определенная кнопка может быть недоступной, а при выборе

из списка может измениться содержимое поля ввода. И наоборот, ввод текста в некоторое поле может автоматически привести к выбору одного или нескольких элементов списка. Если в поле ввода присутствует какой-то текст, то могут быть активизированы кнопки, позволяющие произвести определенное действие над этим текстом (например, изменить либо удалить объект, на который он ссылается).

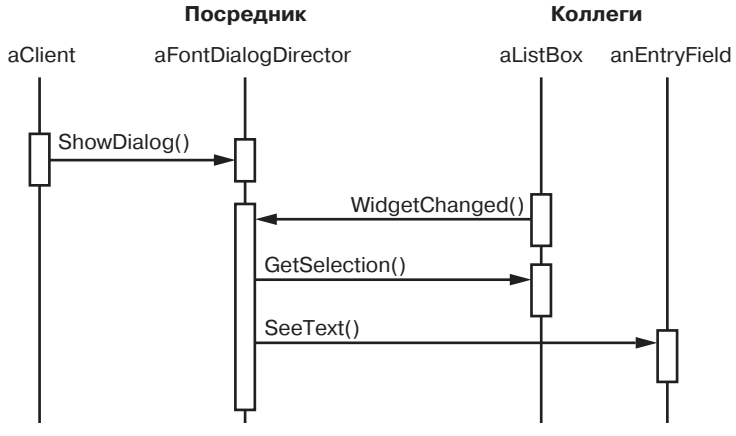
В разных диалоговых окнах зависимости между виджетами могут быть различными. Поэтому, несмотря на то что во всех окнах встречаются однотипные виджеты, просто взять и повторно использовать готовые классы виджетов не удастся, придется производить настройку с целью учета зависимостей. Индивидуальная настройка каждого виджета — утомительное занятие, ибо участвующих классов слишком много.

Всех этих проблем можно избежать, если инкапсулировать коллективное поведение в отдельном объекте-посреднике. Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.

Так, класс `FontDialogDirector` может служить посредником между виджетами в диалоговом окне. Объект этого класса «знает» обо всех виджетах в окне и координирует взаимодействие между ними, то есть выполняет функции центра коммуникаций.



На следующей схеме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка.

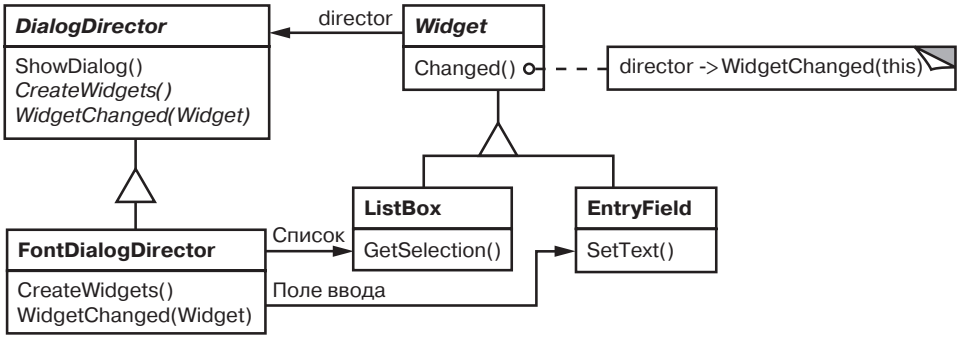


Последовательность событий, в результате которых информация о выбранном элементе списка передается в поле ввода, выглядит так:

1. Список сообщает распорядителю о происшедших в нем изменениях.
2. Распорядитель получает от списка выбранный элемент.
3. Распорядитель передает выбранный элемент полю ввода.
4. Теперь, когда поле ввода содержит какую-то информацию, распорядитель активизирует кнопки для выполнения определенного действия (например, замены обычного шрифта на полужирный или курсив).

Обратите внимание на то, как распорядитель осуществляет посредничество между списком и полем ввода. Виджеты общаются друг с другом не напрямую, а только косвенно через распорядителя. Им вообще не нужно владеть информацией друг о друге, они осведомлены лишь о существовании распорядителя. А поскольку поведение локализовано в одном классе, то его несложно модифицировать или полностью заменить посредством расширения или замены этого класса.

Абстракцию `FontDialogDirector` можно было бы интегрировать в библиотеку классов так, как показано на схеме.



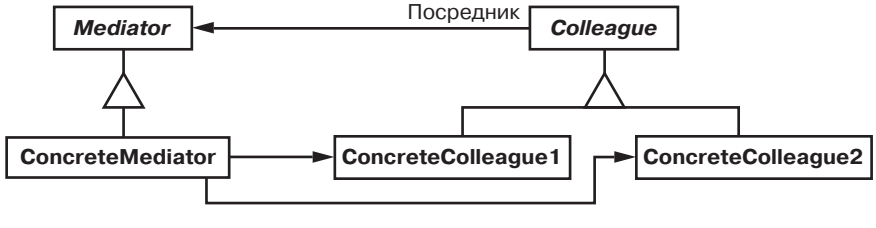
`DialogDirector` — это абстрактный класс, который определяет поведение диалогового окна в целом. Клиенты вызывают его операцию `ShowDialog` для отображения окна на экране. `CreateWidgets` — это абстрактная операция для создания виджетов в диалоговом окне. `WidgetChanged` — еще одна абстрактная операция; с ее помощью виджеты сообщают распорядителю об изменениях. Подклассы `DialogDirector` замещают операции `CreateWidgets` (для создания нужных виджетов) и `WidgetChanged` (для обработки извещений об изменениях).

Применимость

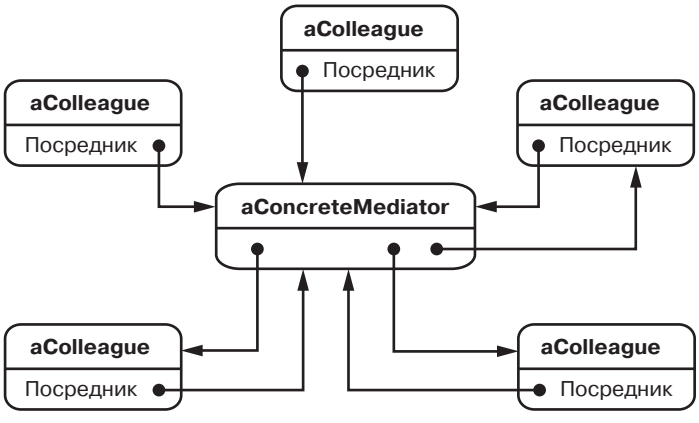
Основные условия для применения паттерна посредник:

- существование объектов, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
- повторное использование объекта затруднено, поскольку он обменивается информацией со многими другими объектами;
- поведение, распределенное между несколькими классами, должно настраиваться без порождения множества подклассов.

Структура



Типичная структура объектов может выглядеть так:



Участники

- **Mediator** (`DialogDirector`) — посредник:
  - определяет интерфейс для обмена информацией с объектами `Colleague`;
- **ConcreteMediator** (`FontDialogDirector`) — конкретный посредник:
  - реализует кооперативное поведение, координируя действия объектов `Colleague`;
  - владеет информацией о коллегах и подсчитывает их;
- Классы **Colleague** (`ListBox`, `EntryField`) — коллеги:
  - каждый класс `Colleague` знает свой объект `Mediator`;
  - все коллеги обмениваются информацией только с посредником во всех случаях, когда ему пришлось бы напрямую взаимодействовать с другими объектами.

Отношения

Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

Результаты

Основные достоинства и недостатки паттерна посредник:

- *снижение числа порождаемых подклассов*. Посредник локализует поведение, которое в противном случае пришлось бы распределять между не-

сколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника `Mediator`, классы коллег `Colleague` можно использовать повторно без каких бы то ни было изменений;

- *ослабление связей между коллегами*. Посредник обеспечивает слабую связанность коллег. Изменять классы `Colleague` и `Mediator` можно независимо друг от друга;
- *упрощение протоколов взаимодействия объектов*. Посредник заменяет взаимодействия «все со всеми» взаимодействиями «один со всеми», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения;
- *абстрагирование способа кооперирования объектов*. Выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это способствует прояснению имеющихся в системе взаимодействий;
- *централизация управления*. Паттерн посредник заменяет сложность взаимодействия сложностью класса-посредника. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник превращается монолит, который трудно сопровождать.

Реализация

При реализации посредника следует обратить внимание на следующие аспекты:

- *избавление от абстрактного класса Mediator*. Если коллеги работают только с одним посредником, то нет необходимости определять абстрактный класс `Mediator`. Обеспечиваемая классом `Mediator` абстракция позволяет коллегам работать с разными подклассами класса `Mediator` и наоборот;
- *обмен информацией между коллегами и посредником*. Коллеги должны обмениваться информацией со своим посредником только тогда, когда возникает представляющее интерес событие. Одним из подходов к реализации посредника является применение паттерна наблюдатель (339). Тогда классы коллег действуют как субъекты, посылающие извещения посреднику о любом изменении своего состояния. Посредник реагирует на них, сообщая об этом другим коллегам.

  При другом подходе в классе `Mediator` определяется специализированный интерфейс уведомления, который позволяет коллегам обме-

ниваться информацией более свободно. В Smalltalk/V для Windows применяется разновидность делегирования: при взаимодействии с посредником коллега передает себя в аргументе, давая посреднику возможность идентифицировать отправителя. Об этом подходе рассказывается в разделе «Пример кода», а о реализации в Smalltalk/V — в разделе «Известные применения».

Пример кода

Для создания диалогового окна, обсуждавшегося в разделе «Мотивация», воспользуемся классом `DialogDirector`. Абстрактный класс `DialogDirector` определяет интерфейс распорядителей:

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

`Widget` — абстрактный базовый класс для всех виджетов. Он располагает информацией о своем распорядителе:

```
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
```

`Changed` вызывает операцию распорядителя `WidgetChanged`. С ее помощью виджеты информируют своего распорядителя о происшедших с ними изменениях:

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

В подклассах DialogDirector переопределена операция WidgetChanged для воздействия на нужные виджеты. Виджет передает ссылку на самого себя в аргументе WidgetChanged, чтобы распорядитель имел информацию об изменившемся виджете. Подклассы DialogDirector переопределяют исключительно виртуальную функцию CreateWidgets для размещения в диалоговом окне нужных виджетов.

ListBox, EntryField и Button — это подклассы Widget для специализированных элементов интерфейса. В классе ListBox есть операция GetSelection для получения текущего множества выделенных элементов, а в классе EntryField — операция SetText для размещения текста в поле ввода:

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

Операция Changed вызывается при нажатии кнопки Button (простой виджет). Это происходит в операции обработки событий мыши HandleMouse:

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
```

Класс FontDialogDirector является посредником между всеми виджетами в диалоговом окне. FontDialogDirector — это подкласс класса DialogDirector:

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

FontDialogDirector отслеживает все виджеты, которые ранее поместил в диалоговое окно. Переопределенная в нем операция CreateWidgets создает виджеты и инициализирует ссылки на них:

```
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // Поместить в список названия шрифтов

    // Разместить все виджеты в диалоговом окне
}
```

Операция WidgetChanged обеспечивает правильную совместную работу виджетов:

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // Изменить шрифт и уничтожить диалоговое окно
        // ...

    } else if (theChangedWidget == _cancel) {
        // закрыть диалог
    }
}
```

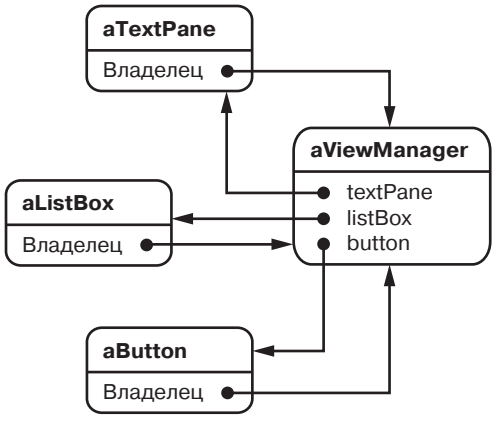
Сложность операции `WidgetChanged` возрастает пропорционально сложности диалогового окна. Конечно, создание очень больших диалоговых окон нежелательно по другим причинам, но в других ситуациях сложность посредника может свести на нет его преимущества.

Известные применения

И в ET++ [WGM88], и в библиотеке классов THINK C [Sym93b] применяются похожие на нашего распорядителя объекты для реализации посредничества между виджетами в диалоговых окнах.

Архитектура приложения в Smalltalk/V для Windows основана на структуре посредника [LaL94]. В этой среде приложение состоит из окна `Window`, которое содержит набор панелей. В библиотеке есть несколько предопределенных объектов-панелей `Pane`, например: `TextPane`, `ListBox`, `Button` и т. д. Их можно использовать без подклассов. Разработчик приложения порождает подклассы только от класса `ViewManager` (диспетчер видов), отвечающего за обмен информацией между панелями. `ViewManager` — это посредник, каждая панель знает своего диспетчера, который считается «владельцем» панели. Панели не ссылаются друг на друга напрямую.

На изображенной схеме объектов показан снимок работающего приложения на стадии выполнения.



В Smalltalk/V для обмена информацией между объектами `Pane` и `ViewManager` используется механизм событий. Панель генерирует событие для получения данных от своего посредника или уведомления его о чем-то важном. С каж-

дым событием связан символ (например, `#select`), который однозначно его идентифицирует. Диспетчер видов регистрирует вместе с панелью селектор метода, который является обработчиком события.

Из следующего фрагмента кода видно, как объект `ListPane` создается внутри подкласса `ViewManager` и как `ViewManager` регистрирует обработчик события `#select`:

```
self addSubpane: (ListPane new
    paneName: 'myListPane';
    owner: self;
    when: #select perform: #listSelect:).
```

При координации сложных обновлений также требуется паттерн посредник. Примером может служить класс `ChangeManager`, упомянутый в описании паттерна наблюдатель (339). Этот класс осуществляет посредничество между субъектами и наблюдателями, чтобы не делать лишних обновлений. Когда объект изменяется, он извещает `ChangeManager`, который координирует обновление и информирует все необходимые объекты.

Аналогичным образом посредник применяется в графических редакторах Unidraw [VL90], где используется класс `CSolver`, следящий за соблюдением ограничений связанности между коннекторами. Объекты в графических редакторах могут быть визуально соединены между собой различными способами. Коннекторы полезны в приложениях, которые автоматически поддерживают связанность, например в редакторах диаграмм и в системах проектирования электронных схем. Класс `CSolver` является посредником между коннекторами. Он преодолевает ограничения связанности и обновляет позиции коннекторов, так чтобы отразить изменения.

Родственные паттерны

Фасад (221) отличается от посредника тем, что он абстрагирует некоторую подсистему объектов для предоставления более удобного интерфейса. Его протокол однонаправленный, то есть объекты фасада направляют запросы классам подсистемы, но не наоборот. Посредник же обеспечивает совместное поведение, которое объекты-коллеги не могут или не «хотят» реализовывать, и его протокол двунаправленный.

Коллеги могут обмениваться информацией с посредником с помощью паттерна наблюдатель (339).

ПАТТЕРН MEMENTO (ХРАНИТЕЛЬ)

Название и классификация паттерна

Хранитель — паттерн поведения объектов.

Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние, так чтобы позднее можно было восстановить в нем объект.

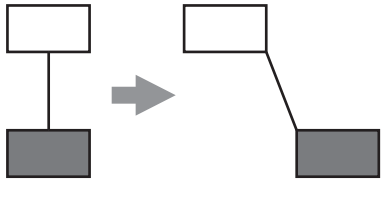
Другие названия

Token (лексема).

Мотивация

Иногда требуется тем или иным способом зафиксировать внутреннее состояние объекта. Такая потребность возникает, например, при реализации контрольных точек и механизмов отмены, позволяющих пользователю отменить пробную операцию или восстановить состояние после ошибки. Его необходимо где-то сохранить, чтобы позднее восстановить в нем объект. Но обычно объекты инкапсулируют все свое состояние полностью или частично, делая его недоступным для других объектов, так что внешнее сохранение состояния невозможно. Раскрытие же состояния нарушило бы принцип инкапсуляции и поставило бы под угрозу надежность и расширяемость приложения.

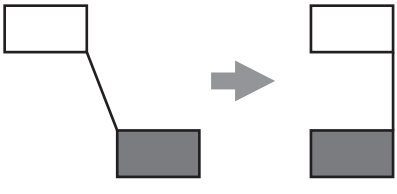
Рассмотрим, например, графический редактор с возможностью связывания объектов. Пользователь может соединить два прямоугольника линией, и они останутся соединенными при любых перемещениях. Редактор сам перерисовывает линию, сохраняя связанность конфигурации.



Система разрешения ограничений — хорошо известный способ поддержания связанности между объектами. Ее функции могут выполняться объектом

класса `ConstraintSolver`, который регистрирует вновь создаваемые соединения и генерирует описывающие их математические уравнения. А когда пользователь каким-то образом модифицирует диаграмму, объект решает эти уравнения. На основании результатов вычислений объект `ConstraintSolver` перерисовывает графику так, чтобы были сохранены все соединения.

Поддержка отмены операций в приложениях не так проста, как может показаться на первый взгляд. Очевидный способ отменить операцию перемещения — сохранение расстояния между старым и новым положением с последующим перемещением объекта на такое же расстояние назад. Однако такое решение не гарантирует, что все объекты окажутся в исходных местах. Допустим, в размещении соединительной линии есть некоторая неопределенность; тогда простое перемещение прямоугольника на прежнее место может не привести к желаемому эффекту.



В общем случае открытого интерфейса `ConstraintSolver` может быть недостаточно для точной отмены всех изменений смежных объектов. Механизм отмены должен работать в тесном взаимодействии с `ConstraintSolver` для восстановления предыдущего состояния, но необходимо также позаботиться о том, чтобы внутренние подробности `ConstraintSolver` не были доступны этому механизму.

Паттерн хранитель поможет решить данную проблему. Хранитель — это объект, в котором сохраняется внутреннее состояние другого объекта — хозяина хранителя. Для работы механизма отмены нужно, чтобы хозяин предоставил хранитель, когда возникнет необходимость записать контрольную точку состояния хозяина. Только хозяину разрешено помещать в хранитель информацию и извлекать ее оттуда, для других объектов хранитель непрозрачен.

В примере графического редактора, который обсуждался выше, в роли хозяина может выступать объект `ConstraintSolver`. Процесс отмены характеризуется следующей последовательностью событий:

1. Редактор запрашивает хранитель у объекта `ConstraintSolver` в процессе выполнения операции перемещения.

2. `ConstraintSolver` создает и возвращает хранитель, в данном случае экземпляр класса `SolverState`. Хранитель `SolverState` содержит структуры данных, описывающие текущее состояние внутренних уравнений и переменных `ConstraintSolver`.
3. Позже, когда пользователь отменяет операцию перемещения, редактор возвращает `SolverState` объекту `ConstraintSolver`.
4. Основываясь на информации, которая хранится в объекте `SolverState`, `ConstraintSolver` изменяет свои внутренние структуры, возвращая формулы и переменные в первоначальное состояние.

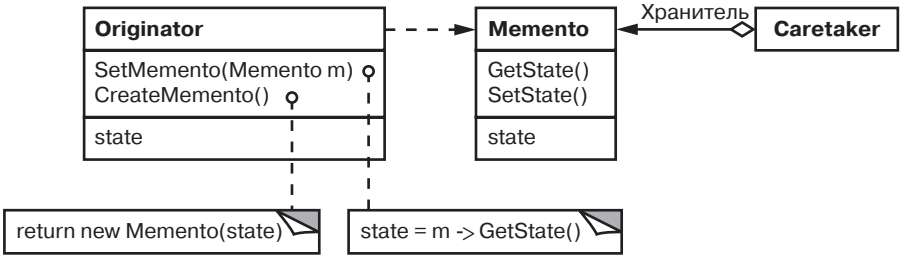
Такая организация позволяет объекту `ConstraintSolver` доверить другим объектам информацию, необходимую для возврата в предыдущее состояние, не раскрывая в то же время свою внутреннюю структуру и представление.

Применимость

Основные условия для применения паттерна хранитель:

- необходимость сохранения снимка состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии, *и*
- прямой интерфейс для получения этого состояния привел бы к раскрытию подробностей реализации и нарушению инкапсуляции объекта.

Структура



Участники

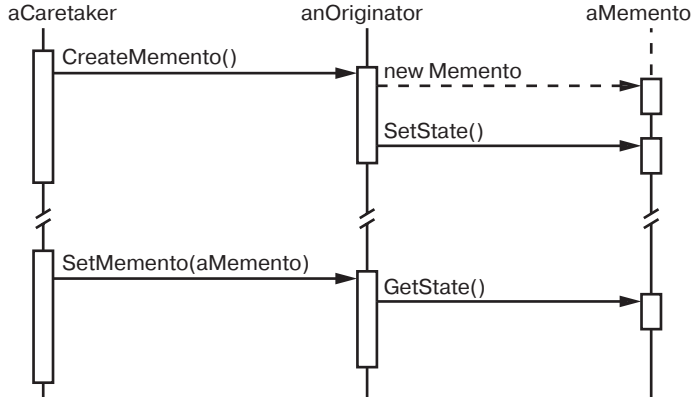
- **Memento** (`SolverState`) — хранитель:
  - сохраняет внутреннее состояние объекта `Originator`. Объем сохраняемой информации может быть различным и определяется потребностями хозяина;

- запрещает доступ всем другим объектам, кроме хозяина. По существу, у хранителей есть два интерфейса. «Посыльный» `Caretaker` видит лишь «узкий» интерфейс хранителя — он может только передавать хранитель другим объектам. Напротив, хозяину доступен «широкий» интерфейс, который обеспечивает доступ ко всем данным, необходимым для восстановления в прежнем состоянии. Идеальный вариант — когда только хозяину, создавшему хранитель, открыт доступ к внутреннему состоянию последнего;

- **Originator** (`ConstraintSolver`) — хозяин:
  - создает хранитель, содержащий снимок текущего внутреннего состояния;
  - использует хранитель для восстановления внутреннего состояния;
- **Caretaker** (механизм отката) — посыльный:
  - отвечает за сохранение хранителя;
  - никогда не выполняет операции с хранителем и не анализирует его внутреннее содержимое.

Отношения

- посыльный запрашивает хранитель у хозяина, некоторое время держит его у себя, а затем возвращает хозяину, как показано на следующей диаграмме взаимодействий.



Иногда этого не происходит, так как последнему не нужно восстанавливать прежнее состояние;

- хранители пассивны. Только хозяин, создавший хранитель, имеет доступ к информации о состоянии.

Результаты

Основные достоинства и недостатки паттерна хранитель:

- *сохранение границ инкапсуляции*. Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн изолирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции;
- *упрощение структуры хозяина*. При других вариантах дизайна, сохраняющего границы инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу;
- *потенциальные затраты при использовании хранителей*. С хранителями могут быть связаны заметные затраты, если хозяин должен копировать большой объем информации для сохранения хранителя в памяти, или если клиенты создают и возвращают хранители достаточно часто. Если затраты на инкапсуляцию и восстановление состояния хозяина велики, то этот паттерн не всегда подходит (см. также обсуждение инкрементности в разделе «Реализация»);
- *определение «узкого» и «широкого» интерфейсов*. В некоторых языках сложно гарантировать, что только хозяин может получить доступ к состоянию хранителя;
- *скрытая плата за содержание хранителя*. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Следовательно, нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Реализация

При рассмотрении паттерна хранитель следует обратить внимание на следующие аспекты:

- *языковая поддержка*. У хранителей есть два интерфейса: «широкий» для хозяев и «узкий» для всех остальных объектов. В идеале язык реализации должен поддерживать два уровня статического контроля доступа. В C++ это возможно, если объявить хозяина другом хранителя и сделать закрытым «широкий» интерфейс последнего (с помощью

ключевого слова `private`). Открытым (`public`) остается только «узкий» интерфейс. Пример:

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;
    // Внутренние структуры данных
    // ...
};

class Memento {
public:
    // Узкий открытый интерфейс
    virtual ~Memento();
private:
    // Закрытые члены, доступные только хозяину Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```

- *сохранение инкрементных изменений.* Если хранители создаются и возвращаются своему хозяину в предсказуемой последовательности, то хранитель может сохранить лишь *инкрементные изменения* во внутреннем состоянии хозяина.

  Например, допускающие отмену команды в списке истории могут пользоваться хранителями для восстановления первоначального состояния (см. описание паттерна команда (275)). Список истории предназначен только для отмены и повтора команд. Это означает, что хранители могут работать лишь с изменениями, сделанными командой, а не с полным состоянием объекта. В примере из раздела «Мотивация» объект, отменяющий ограничения, может хранить только такие внутренние структуры, которые изменяются с целью сохранить линию, соединяющую прямоугольники, а не абсолютные позиции всех объектов.

Пример кода

Приведенный пример кода на языке C++ иллюстрирует рассмотренный выше пример класса `ConstraintSolver` для разрешения ограничений. Мы используем объекты `MoveCommand` (см. паттерн команда (275)) для выполнения и отмены переноса графического объекта из одного места в другое. Графический редактор вызывает операцию `Execute` объекта-команды, чтобы переместить объект, и команду `Unexecute`, чтобы отменить перемещение. В команде хранятся координаты места назначения, величина смещения и экземпляр класса `ConstraintSolverMemento` — хранителя, содержащего состояние объекта `ConstraintSolver`:

```
class Graphic;
    // Базовый класс графических объектов

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

Ограничения связанности устанавливаются классом `ConstraintSolver`. Его основная функция `Solve` обрабатывает ограничения, регистрируемые операцией `AddConstraint`. Для поддержки отмены действий состояние объекта `ConstraintSolver` можно сохранить в экземпляре класса `ConstraintSolverMemento` с помощью операции `CreateMemento`. В предыдущее состояние объект `ConstraintSolver` возвращается вызовом `SetMemento`. `ConstraintSolver` является примером паттерна одиночка (157):

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );

    ConstraintSolverMemento* CreateMemento();
```

```
    void SetMemento(ConstraintSolverMemento*);
private:
    // Нетривиальное состояние и операции
    // для поддержки семантики связанности
};
class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // Закрытое состояние Solver
};
```

С такими интерфейсами можно реализовать функции `Execute` и `Unexecute` в классе `MoveCommand` следующим образом:

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento();
    // Создание хранителя
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state);
    // Восстановление состояния
    solver->Solve();
}
```

`Execute` запрашивает хранителя `ConstraintSolverMemento` перед началом перемещения графического объекта. `Unexecute` возвращает объект на прежнее место, восстанавливает состояние `Solver` и обращается к последнему с целью отменить ограничения.

Известные применения

Предыдущий пример основан на поддержке связанности в каркасе Unidraw с помощью класса `CSolver` [VL90].

В коллекциях языка Dylan [App92] для итерации предусмотрен интерфейс, напоминающий паттерн хранитель. Для этих коллекций существует понятие состояния объекта, которое является хранителем, представляющим состояние итерации. Представление текущего состояния каждой коллекции может

быть любым, но оно полностью скрыто от клиентов. Решение, используемое в языке Dylan, можно написать на C++ следующим образом:

```
template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next(IterationState*);
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item&);
    void Remove(const Item&);
    // ...
};
```

Операция `CreateInitialState` возвращает инициализированный объект `IterationState` для коллекции. Операция `Next` переходит к следующему объекту в порядке итерации, фактически она увеличивает на единицу индекс итерации. Операция `IsDone` возвращает `true`, если в результате выполнения `Next` мы оказались за последним элементом коллекции. Операция `CurrentItem` разыменовывает объект состояния и возвращает тот элемент коллекции, на который он ссылается. `Copy` возвращает копию данного объекта состояния. Это имеет смысл, когда необходимо оставить закладку в некотором месте, пройденном во время итерации.

Для заданного класса `ItemType` обход коллекции, составленной из его экземпляров, может выполняться так[1]:

```
class ItemType {
public:
    void Process();
    // ...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();
```

[1] Отметим, что в нашем примере объект состояния удаляется по завершении итерации. Но оператор delete не будет вызван, если `ProcessItem` возбудит исключение, поэтому в памяти остается мусор. Это проблема в языке C++, но не в Dylan, где есть сборщик мусора. Решение проблемы обсуждается на с. 258.

```
while (!aCollection.IsDone(state)) {
    aCollection.CurrentItem(state)->Process();
    aCollection.Next(state);
}
delete state;
```

У интерфейса итерации, основанного на паттерне хранитель, есть два преимущества:

- с одной коллекцией может быть связано несколько активных состояний (как и в случае с паттерном итератор (302));
- поддержка итерации не требует нарушения инкапсуляции коллекции. Хранитель интерпретируется только самой коллекцией, больше никто к нему доступа не имеет. При других подходах приходится нарушать инкапсуляцию, объявляя классы итераторов друзьями классов коллекций (см. описание паттерна итератор (302)). В случае с хранителем ситуация противоположная: класс коллекции `Collection` является другом класса `IteratorState`.

В библиотеке QOCA для разрешения ограничений в хранителях содержится информация об изменениях. Клиент может получить хранитель, характеризующий текущее решение системы ограничений. В хранителе находятся только те переменные ограничений, которые были преобразованы со времени последнего решения. Обычно при каждом новом решении изменяется лишь небольшое подмножество переменных `Solver`. Но этого достаточно, чтобы вернуть `Solver` к предыдущему решению; для отката к более ранним решениям необходимо иметь все промежуточные хранители. Поэтому передавать хранители в произвольном порядке нельзя; QOCA использует механизм ведения истории для возврата к прежним решениям.

Родственные паттерны

Команда (275): команды помещают информацию о состоянии, необходимую для отмены выполненных действий, в хранители.

Итератор (302): хранители могут использоваться для выполнения итераций, как было показано выше.

ПАТТЕРН OBSERVER (НАБЛЮДАТЕЛЬ)

Название и классификация паттерна

Наблюдатель — паттерн поведения объектов.

Назначение

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

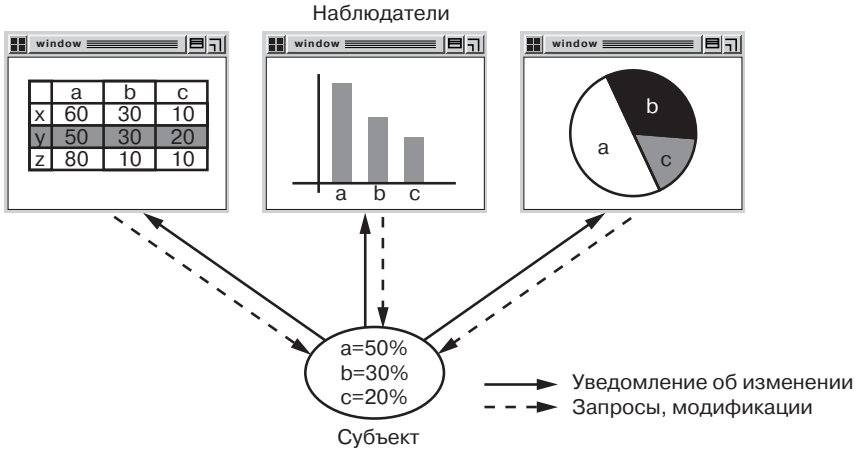
Другие названия

Dependents (подчиненные), Publish-Subscribe (издатель — подписчик).

Мотивация

Одним из типичных побочных эффектов разбиения системы на взаимодействующие классы является необходимость согласования состояния взаимосвязанных объектов. Однако согласованность не должна достигаться за счет жесткой связанности классов, так как это снижает возможности их повторного использования.

Например, во многих библиотеках для построения графических интерфейсов пользователя презентационные аспекты интерфейса отделены от данных приложения [KP88, LVC89, P+88, WGM88]. С классами, описывающими данные и их представление, можно работать автономно; при этом они могут работать и совместно. Электронная таблица и объект-диаграмма не имеют информации друг о друге, поэтому их можно использовать по отдельности. Но *ведут* они себя так, как будто знают друг о друге. Когда пользователь изменяет информацию в таблице, все изменения немедленно отражаются на диаграмме, и наоборот.



При таком поведении подразумевается, что и электронная таблица, и диаграмма зависят от данных объекта и поэтому должны уведомляться о лю-

бых изменениях в его состоянии. И нет никаких причин, ограничивающих количество зависимых объектов; для работы с одними и теми же данными может существовать любое число пользовательских интерфейсов.

Паттерн наблюдатель описывает, как устанавливаются такие отношения. Ключевыми объектами в нем являются субъект и наблюдатель. У субъекта может быть сколько угодно зависимых от него наблюдателей. Все наблюдатели уведомляются об изменениях в состоянии субъекта. Получив уведомление, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним свое состояние.

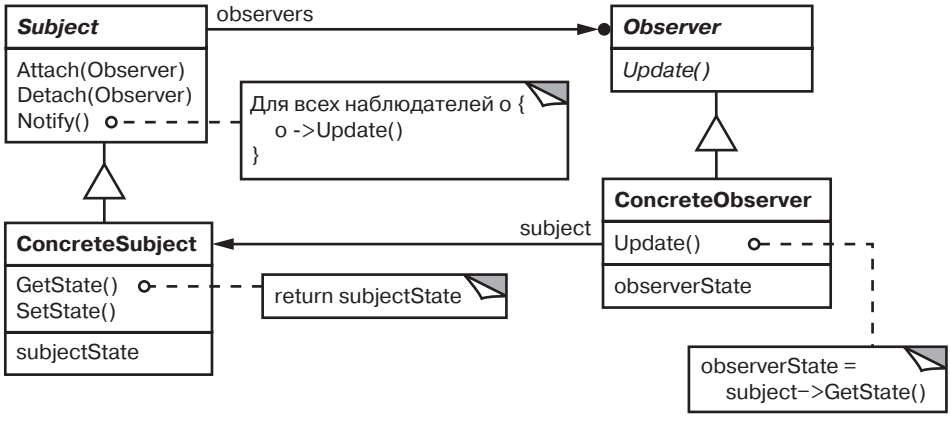
Такого рода взаимодействие часто называется отношением издатель — подписчик. Субъект издает или публикует уведомления и рассылает их, даже не имея информации о том, какие объекты являются подписчиками. На получение уведомлений может подписаться неограниченное количество наблюдателей.

Применимость

Основные условия для применения паттерна наблюдатель:

- у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;
- при модификации одного объекта требуется изменить другие, и вы не знаете, сколько именно объектов нужно изменить;
- один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, объекты не должны быть тесно связаны между собой.

Структура



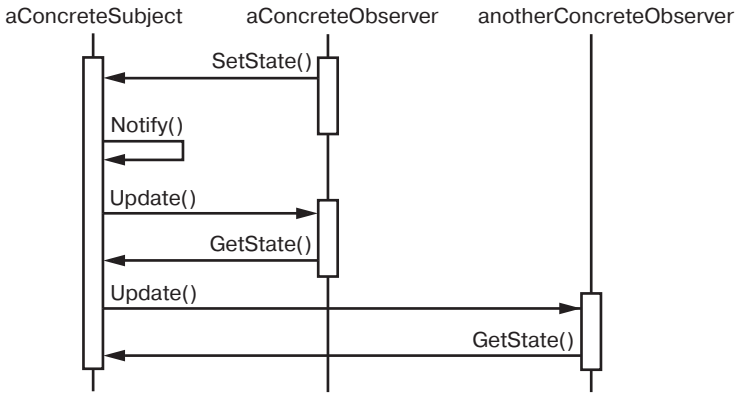
Участники

- **Subject** — субъект:
  - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
  - предоставляет интерфейс для присоединения и отделения наблюдателей;
- **Observer** — наблюдатель:
  - определяет интерфейс обновления для объектов, которые должны уведомляться об изменении субъекта;
- **ConcreteSubject** — конкретный субъект:
  - сохраняет состояние, представляющее интерес для конкретного наблюдателя `ConcreteObserver`;
  - посылает информацию своим наблюдателям, когда происходит изменение;
- **ConcreteObserver** — конкретный наблюдатель:
  - хранит ссылку на объект класса `ConcreteSubject`;
  - сохраняет данные, которые должны быть согласованы с данными субъекта;
  - реализует интерфейс обновления, определенный в классе `Observer`, чтобы поддерживать согласованность с субъектом.

Отношения

- объект `ConcreteSubject` уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;
- после получения от конкретного субъекта уведомления об изменении объект `ConcreteObserver` может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта.

  На схеме взаимодействий показаны отношения между субъектом и двумя наблюдателями.



Результаты

Паттерн наблюдатель позволяет изменять субъекты и наблюдатели независимо друг от друга. Субъекты разрешается повторно использовать без участия наблюдателей, и наоборот. Это дает возможность добавлять новых наблюдателей без модификации субъекта или других наблюдателей.

Основные достоинства и недостатки паттерна наблюдатель:

- *абстрактная связанность субъекта и наблюдателя.* Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса `Observer`. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму.

  Поскольку субъект и наблюдатель не являются тесно связанными, они могут находиться на разных уровнях абстракции системы. Субъект более низкого уровня может уведомлять наблюдателей, находящихся на верхних уровнях, не нарушая иерархии системы. Если бы субъект и наблюдатель представляли собой единое целое, то получающийся объект либо пересекал бы границы уровней (нарушая принцип их формирования), либо должен был находиться на каком-то одном уровне (нарушая абстракцию уровня);

- *поддержка широковещательных коммуникаций.* В отличие от обычного запроса, для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекта не интересует, сколько существует таких объектов; от него требуется всего лишь уведомить сво-

их наблюдателей. Таким образом, мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его;

- *неожиданные обновления*. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная на первый взгляд операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов. Более того, нечетко определенные или плохо поддерживаемые критерии зависимости могут стать причиной непредвиденных обновлений, отследить которые очень сложно.

  Проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что *именно* изменилось в субъекте. Без дополнительного протокола, который позволяет получить информацию об изменениях, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Реализация

В этом разделе обсуждаются вопросы, относящиеся к реализации механизма зависимостей:

- *связывание субъектов с наблюдателями*. Этим простейшим способом субъект может отслеживать всех наблюдателей, которым он должен посылать уведомления — то есть хранить на них явные ссылки. Однако при большом числе субъектов при нескольких наблюдателях это может привести к слишком высоким затратам. Один из возможных компромиссов — экономия памяти за счет времени с использованием ассоциативного массива (например, хеш-таблицы) для хранения отображения между субъектами и наблюдателями. Тогда субъект, у которого нет наблюдателей, не будет зря расходовать память. С другой стороны, при таком подходе увеличивается время поиска наблюдателей;

- *наблюдение более чем за одним субъектом*. Иногда наблюдатель может зависеть более чем от одного субъекта. Например, у электронной таблицы бывает более одного источника данных. В таких случаях необходимо расширить интерфейс `Update`, чтобы наблюдатель мог узнать, какой субъект прислал уведомление. Субъект может просто передать себя в параметре операции `Update`, тем самым сообщая наблюдателю, что именно нужно обследовать;

- *кто инициирует обновление?* Для сохранения согласованности субъект и его наблюдатели полагаются на механизм уведомлений. Но какой

именно объект вызывает операцию `Notify` для инициирования обновления? Возможны два варианта:

- операции класса `Subject`, изменившие состояние, вызывают `Notify` для уведомления об этом изменении. Преимущество такого подхода в том, что клиентам не надо помнить о необходимости вызывать операцию `Notify` субъекта. Недостаток же заключается в том, что при выполнении каждой из нескольких последовательных операций будут проводиться обновления, что может привести к неэффективной работе программы;
- ответственность за своевременный вызов `Notify` возлагается на клиента. Преимущество: клиент может отложить инициирование обновления до завершения серии изменений, исключив тем самым ненужные промежуточные обновления. Недостаток: у клиентов появляется дополнительная обязанность. Это увеличивает вероятность ошибок, поскольку клиент может забыть вызвать `Notify`;

- *висячие ссылки на удаленных субъектов*. Удаление субъекта не должно приводить к появлению висячих ссылок у наблюдателей. Избежать этого можно, например, поручив субъекту уведомлять всех своих наблюдателей о своем удалении, чтобы они могли уничтожить хранимые у себя ссылки. В общем случае простое удаление наблюдателей не годится, так как на них могут ссылаться другие объекты, и под их наблюдением могут находиться другие субъекты;
- *гарантии целостности состояния субъекта перед отправкой уведомления*. Важно быть уверенным, что перед вызовом операции `Notify` состояние субъекта непротиворечиво, поскольку в процессе обновления собственного состояния наблюдатели будут опрашивать состояние субъекта.

  Правило непротиворечивости легко случайно нарушить, если операции одного из подклассов класса `Subject` вызывают унаследованные операции. Например, в следующем фрагменте уведомление отправляется, когда состояние субъекта противоречиво:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
        // Отправить уведомление

    _myInstVar += newValue;
        // Обновить состояние подкласса (слишком поздно!)
}
```

Этой ловушки можно избежать, отправляя уведомления из шаблонных методов (см. описание паттерна шаблонный метод (373)) абстрактного класса `Subject`. Определите примитивную операцию, замещаемую в подклассах, и обратитесь к `Notify`, используя последнюю операцию в шаблонном методе. В таком случае существует гарантия, что состояние объекта непротиворечиво, если операции `Subject` замещены в подклассах:

```
void Text::Cut (TextRange r) {
    ReplaceRange(r);       // Переопределяется в подклассах
    Notify();
}
```

Кстати, всегда желательно фиксировать, какие операции класса `Subject` инициируют обновления;

- *предотвращение зависимости протокола обновления от наблюдателя*: модели вытягивания и проталкивания. В реализациях паттерна наблюдатель субъект довольно часто транслирует всем подписчикам дополнительную информацию о характере изменения. Она передается в виде аргумента операции `Update`, и объем ее меняется в широких диапазонах.

  На одном полюсе находится так называемая *модель проталкивания* (push model), когда субъект посылает наблюдателям детальную информацию об изменении независимо от того, нужно ли им это. На другом — *модель вытягивания* (pull model), когда субъект не посылает ничего, кроме минимального уведомления, а наблюдатели запрашивают детали позднее.

  Модель вытягивания подчеркивает неинформированность субъекта о своих наблюдателях, а в модели проталкивания предполагается, что субъект владеет определенной информацией о потребностях наблюдателей. В случае применения модели проталкивания степень повторного их использования может снизиться, так как классы `Subject` делают предположения о классах `Observer`, которые могут оказаться неправильными. С другой стороны, модель вытягивания может оказаться неэффективной, ибо наблюдателям без помощи субъекта необходимо выяснять, что изменилось;

- *явное определение модификаций, представляющих интерес*. Эффективность обновления можно повысить, расширив интерфейс регистрации субъекта, то есть предоставив возможность при регистрации наблюдателя указать, какие именно события его интересуют. Когда событие происходит, субъект информирует лишь тех наблюдателей, которые

проявили к нему интерес. Чтобы получать конкретное событие, наблюдатели присоединяются к своим субъектам следующим образом:

```
void Subject::Attach(Observer*, Aspect& interest);
```

где `interest` определяет представляющее интерес событие. В момент отправки уведомления субъект передает своим наблюдателям изменившийся аспект в виде параметра операции `Update`. Например:

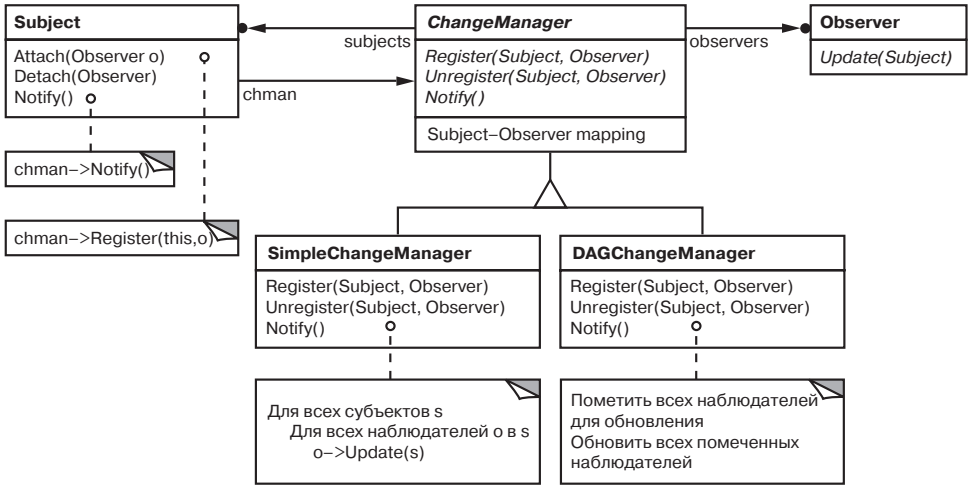
```
void Observer::Update(Subject*, Aspect& interest);
```

- *инкапсуляция сложной семантики обновления*. Если отношения зависимости между субъектами и наблюдателями становятся особенно сложными, то может потребоваться объект, инкапсулирующий эти отношения. Будем называть его `ChangeManager` (менеджер изменений). Он должен свести к минимуму объем работы, необходимой для того, чтобы наблюдатели смогли отразить изменения субъекта. Например, если некоторая операция влечет за собой изменения в нескольких независимых субъектах, то хотелось бы, чтобы наблюдатели уведомлялись после того, как будут модифицированы *все* субъекты, дабы не ставить в известность одного и того же наблюдателя несколько раз.

  У класса `ChangeManager` есть три обязанности:

  - строить отображение между субъектом и его наблюдателями и предоставлять интерфейс для поддержания отображения в актуальном состоянии. Это освобождает субъектов от необходимости хранить ссылки на своих наблюдателей и наоборот;
  - определять конкретную стратегию обновления;
  - обновлять всех зависимых наблюдателей по запросу от субъекта.

  На следующей схеме представлена простая реализация паттерна наблюдатель с использованием менеджера изменений `ChangeManager`. Имеется два специализированных менеджера. `SimpleChangeManager` всегда обновляет всех наблюдателей каждого субъекта, а `DAGChangeManager` обрабатывает направленные ациклические графы зависимостей между субъектами и их наблюдателями. Когда наблюдатель должен «присматривать» за несколькими субъектами, предпочтительнее использовать `DAGChangeManager`. В этом случае изменение сразу двух или более субъектов может привести к избыточным обновлениям. Объект `DAGChangeManager` гарантирует, что наблюдатель в любом случае получит только одно уведомление. Если обновление одного и того же наблюдателя допускается несколько раз подряд, то вполне достаточно объекта `SimpleChangeManager`.



`ChangeManager` — это пример паттерна посредник (319). В общем случае есть только один объект `ChangeManager`, известный всем участникам. Поэтому полезен будет также и паттерн одиночка (157);

- *комбинирование классов Subject и Observer*. В библиотеках классов, которые написаны на языках, не поддерживающих множественного наследования (например, на Smalltalk), обычно не определяются отдельные классы `Subject` и `Observer`. Их интерфейсы комбинируются в одном классе. Это позволяет определить объект, выступающий в роли одновременно субъекта и наблюдателя, без множественного наследования. Так, в Smalltalk интерфейсы `Subject` и `Observer` определены в корневом классе `Object` и потому доступны вообще всем классам.

Пример кода

Интерфейс наблюдателя определен в абстрактном классе `Observer`:

```
class Subject;

class Observer {
public:
    virtual ~ Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

Такая реализация поддерживает несколько субъектов для одного наблюдателя. Передача субъекта в параметре операции `Update` позволяет наблюдателю определить, какой из наблюдаемых им субъектов изменился.

Аналогичным образом в абстрактном классе `Subject` определен интерфейс субъекта:

```
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};
void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

`ClockTimer` — это конкретный субъект, который следит за временем суток. Он оповещает наблюдателей каждую секунду. Класс `ClockTimer` предоставляет интерфейс для получения отдельных компонентов времени: часа, минуты, секунды и т. д.:

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};
```

Операция `Tick` вызывается через одинаковые интервалы внутренним таймером. Тем самым обеспечивается правильный отсчет времени. При этом обновляется внутреннее состояние объекта `ClockTimer` и вызывается операция `Notify` для извещения наблюдателей об изменении:

```
void ClockTimer::Tick () {
    // Обновить внутреннее представление времени
    // ...
    Notify();
}
```

Теперь можно определить класс `DigitalClock` для вывода времени. Свою графическую функциональность он наследует от класса `Widget`, предоставляемого библиотекой для построения пользовательских интерфейсов. Интерфейс наблюдателя примешивается к интерфейсу `DigitalClock` путем наследования от класса `Observer`:

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // Замещает операцию класса Observer

    virtual void Draw();
        // Замещает операцию класса Widget;
        // определяет способ отображения часов
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}
```

Прежде чем начнется рисование часов посредством операции `Update`, будет проверено, что уведомление получено именно от объекта таймера:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
```

```
    }
}

void DigitalClock::Draw () {
    // Получить новые значения от субъекта

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // Нарисовать цифровые часы
}
```

Аналогичным образом определяется класс `AnalogClock`:

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

Следующий код создает объекты классов `AnalogClock` и `DigitalClock`, которые всегда показывают одно и то же время:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

При каждом срабатывании таймера `timer` оба экземпляра часов обновляются и перерисовывают себя.

Известные применения

Первый и, возможно, самый известный пример паттерна наблюдатель появился в схеме «модель/представление/контроллер» (MVC) языка Smalltalk, которая представляет собой каркас для построения пользовательских интерфейсов в среде Smalltalk [KP88]. Класс `Model` в MVC — субъект, а `View` — базовый класс для наблюдателей. В языках Smalltalk, ET++ [WGM88] и библиотеке классов THINK [Sym93b] предлагается общий механизм зависимостей, в котором интерфейсы субъекта и наблюдателя помещены в класс, являющийся общим родителем всех остальных системных классов.

Среди других библиотек для построения интерфейсов пользователя, в которых используется паттерн наблюдатель, стоит упомянуть InterViews [LVC89],

Andrew Toolkit [P+88] и Unidraw [VL90]. В InterViews явно определены классы `Observer` и `Observable` (для субъектов). В библиотеке Andrew они называются *представлением* (view) и *объектом данных* (data object) соответственно. Unidraw делит объекты графического редактора на части `View` (для наблюдателей) и `Subject`.

Родственные паттерны

Посредник (319): класс `ChangeManager` действует как посредник между субъектами и наблюдателями, инкапсулируя сложную семантику обновления.

Одиночка (157): класс `ChangeManager` может воспользоваться паттерном одиночка, чтобы гарантировать уникальность и глобальную доступность менеджера изменений.

ПАТТЕРН STATE (СОСТОЯНИЕ)

Название и классификация паттерна

Состояние — паттерн поведения объектов.

Назначение

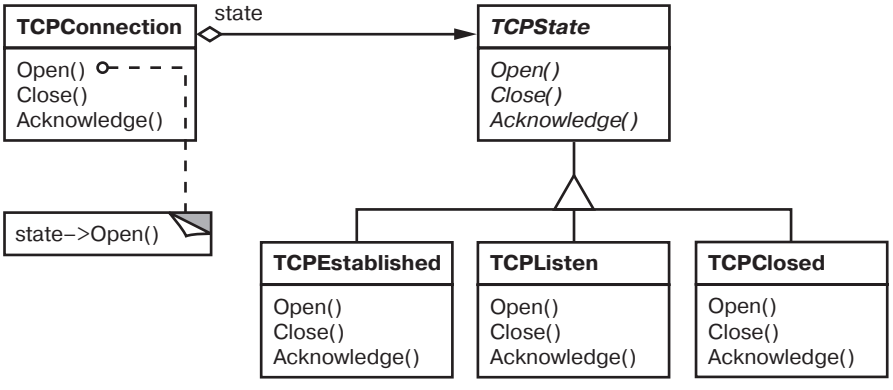
Позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Мотивация

Рассмотрим класс `TCPConnection`, представляющий сетевое соединение. Объект этого класса может находиться в одном из нескольких состояний: `Established` (установлено), `Listening` (прослушивание), `Closed` (закрыто). Когда объект `TCPConnection` получает запросы от других объектов, то в зависимости от текущего состояния он отвечает по-разному. Например, ответ на запрос `Open` (открыть) зависит от того, находится ли соединение в состоянии `Closed` или `Established`. Паттерн состояние описывает, каким образом объект `TCPConnection` может вести себя по-разному, находясь в различных состояниях.

Основная идея этого паттерна заключается в том, чтобы ввести абстрактный класс `TCPState` для представления различных состояний соединения. Этот класс объявляет интерфейс, общий для всех классов, описывающих различные рабочие состояния. В подклассах `TCPState` реализовано по-

ведение, специфичное для конкретного состояния. Например, в классах `TCPEstablished` и `TCPClosed` реализовано поведение, характерное для состояний `Established` и `Closed` соответственно.



Класс `TCPConnection` хранит у себя объект состояния (экземпляр некоторого подкласса `TCPState`), представляющий текущее состояние соединения, и делегирует все зависящие от состояния запросы этому объекту. `TCPConnection` использует свой экземпляр подкласса `TCPState` для выполнения операций, свойственных только данному состоянию соединения.

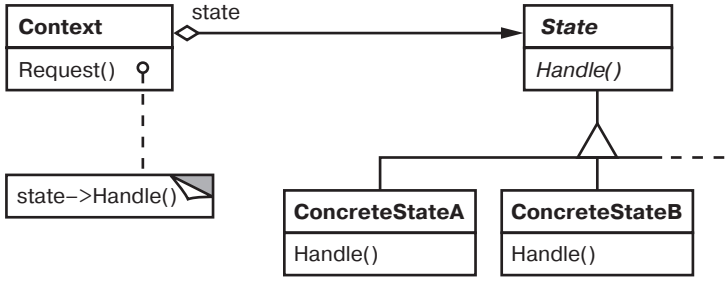
При каждом изменении состояния соединения `TCPConnection` изменяет свой объект-состояние. Например, когда установленное соединение закрывается, `TCPConnection` заменяет экземпляр класса `TCPEstablished` экземпляром `TCPClosed`.

Применимость

Основные условия для применения паттерна состояние:

- поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Структура



Участники

- **Context** (`TCPConnection`) — контекст:
  - определяет интерфейс, представляющий интерес для клиентов;
  - хранит экземпляр подкласса `ConcreteState`, которым определяется текущее состояние;
- **State** (`TCPState`) — состояние:
  - определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста `Context`;
- Подклассы `ConcreteState` (`TCPEstablished`, `TCPListen`, `TCPClosed`) — конкретное состояние:
  - каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста `Context`.

Отношения

- Класс `Context` делегирует зависящие от состояния запросы текущему объекту `ConcreteState`;
- контекст может передать себя в качестве аргумента объекту `State`, который будет обрабатывать запрос. Это дает возможность объекту-состоянию при необходимости получить доступ к контексту;
- `Context` — это основной интерфейс для клиентов. Клиенты могут конфигурировать контекст объектами состояния `State`. Один раз сконфигурировав контекст, клиенты уже не должны напрямую связываться с объектами состояния;
- либо `Context`, либо подклассы `ConcreteState` могут решить, при каких условиях и в каком порядке происходит смена состояний.

Результаты

Результаты использования паттерна состояние:

- *локализация поведения, зависящего от состояния, и деление его на части, соответствующие состояниям.* Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса `State`, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов. Вместо этого можно было бы использовать данные-члены для определения внутренних состояний, тогда операции объекта `Context` проверяли бы эти данные. Но в таком случае похожие условные операторы или операторы ветвления были бы разбросаны по всему коду класса `Context`. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение.

  Паттерн состояние позволяет решить эту проблему, но одновременно порождает другую, поскольку поведение для различных состояний оказывается распределенным между несколькими подклассами `State`. Это увеличивает число классов. Конечно, один класс компактнее, но если состояний много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами.

  Наличие громоздких условных операторов нежелательно, равно как и длинных процедур. Они слишком монолитны, поэтому с модификацией и расширением кода возникают проблемы. Паттерн состояние предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы `if` или `switch`, а распределена между подклассами `State`. При инкапсуляции каждого перехода и действия в класс состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение;

- *явно выраженные переходы между состояниями.* Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты `State` могут защитить контекст `Context` от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста — это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной `State` в классе `Context`), а не нескольких [dCLF93];

- *возможность совместного использования объектов состояния*. Если в объекте состояния `State` отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект `State`. Когда состояния разделяются таким образом, они являются, по сути дела, приспособленцами (см. описание паттерна приспособленец (231)), у которых нет внутреннего состояния, а есть только поведение.

Реализация

При реализации паттерна состояние следует обратить внимание на следующие аспекты:

- *что определяет переходы между состояниями*. Паттерн состояние ничего не сообщает о том, какой участник определяет критерий перехода между состояниями. Если критерии зафиксированы, то их можно реализовать непосредственно в классе `Context`. Однако в общем случае более гибкий и правильный подход заключается в том, чтобы позволить самим подклассам класса State определять следующее состояние и момент перехода. Для этого в класс Context надо добавить интерфейс, позволяющий объектам `State` установить состояние контекста. Такую децентрализованную логику переходов проще модифицировать и расширять — нужно лишь определить новые подклассы `State`. Недостаток децентрализации в том, что каждый подкласс `State` должен знать еще хотя бы об одном подклассе, что вносит реализационные зависимости между подклассами;
- *табличная альтернатива*. Том Каргилл (Tom Cargill) в книге C++ Programming Style [Car92] описывает другой способ структурирования кода, управляемого состояниями. Он использует таблицу для отображения входных данных на переходы между состояниями. С ее помощью можно определить, в какое состояние нужно перейти при поступлении некоторых входных данных. По существу, тем самым мы заменяем условный код (или виртуальные функции, если речь идет о паттерне состояние) поиском в таблице. Основное преимущество таблиц — в их регулярности: для изменения критериев перехода достаточно модифицировать только данные, а не код. Но есть и недостатки:
  - поиск в таблице часто менее эффективен, чем вызов функции (виртуальной);
  - представление логики переходов в однородном табличном формате делает критерии менее явными и, стало быть, усложняет их понимание;
  - обычно трудно добавить действия, которыми сопровождаются переходы между состояниями. Табличный метод учитывает состояния и переходы

между ними, но его необходимо дополнить, чтобы при каждом изменении состояния можно было выполнять произвольные вычисления.

Главное различие между конечными автоматами на базе таблиц и паттерном состояние можно сформулировать так: паттерн состояние моделирует поведение, зависящее от состояния, а табличный метод акцентирует внимание на определении переходов между состояниями;

- *создание и уничтожение объектов состояния.* В процессе разработки обычно приходится выбирать между: (1) созданием объектов состояния, когда в них возникает необходимость, и уничтожением сразу после использования, и (2) созданием их заранее и навсегда.

  Первый вариант предпочтителен в тех случаях, когда возможные состояния системы неизвестны заранее, а контекст изменяет состояние сравнительно редко. При этом объекты, которые никогда не будут использованы, не создаются, что может быть существенно, если в объектах состояния хранится много информации. Если изменения состояния происходят часто, и уничтожать представляющие их объекты было бы нежелательно (ибо они могут очень скоро понадобиться вновь), лучше воспользоваться вторым подходом. Время на создание объектов затрачивается только один раз, в самом начале, а на уничтожение — не затрачивается вовсе. Правда, этот вариант может оказаться неудобным, так как в контексте должны храниться ссылки на все состояния, в которых теоретически может оказаться система;

- *использование динамического наследования.* Изменение поведения по конкретному запросу может достигаться сменой класса объекта во время выполнения, но в большинстве объектно-ориентированных языков такая возможность не поддерживается. Исключение составляет Self [US87] и другие основанные на делегировании языки, которые предоставляют такой механизм и, следовательно, поддерживают паттерн состояние напрямую. Объекты в Self могут делегировать операции другим объектам, обеспечивая тем самым некую форму динамического наследования. Изменение целевого объекта делегирования во время выполнения фактически приводит к изменению структуры графа наследования. Такой механизм позволяет объектам варьировать поведение путем изменения своего класса.

Пример кода

В следующем примере приведен код на языке C++ с TCP-соединением из раздела «Мотивация». Это упрощенный вариант протокола TCP, в нем, конечно же, представлен не весь протокол и даже не все состояния TCP-соединений[1].

[1] Пример основан на описании протокола установления TCP-соединений из книги Линча и Роуза [LR93].

Прежде всего определим класс TCPConnection, который предоставляет интерфейс для передачи данных и обрабатывает запросы на изменение состояния:

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

В переменной _state класса TCPConnection хранится экземпляр класса TCPState. Этот класс дублирует интерфейс изменения состояния, определенный в классе TCPConnection. Каждая операция TCPState получает экземпляр TCPConnection в параметре, что позволяет объекту TCPState получить доступ к данным объекта TCPConnection и изменить состояние соединения:

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

TCPConnection делегирует все запросы, зависящие от состояния, хранимому в _state экземпляру TCPState. Кроме того, в классе TCPConnection существует операция, с помощью которой в эту переменную можно записать указатель

на другой объект `TCPState`. Конструктор класса `TCPConnection` инициализирует `_state` указателем на состояние `TCPClosed` (оно будет определено ниже):

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

В классе `TCPState` реализовано поведение по умолчанию для всех делегированных ему запросов. Он может также изменить состояние объекта `TCPConnection` посредством операции `ChangeState`. `TCPState` объявляется другом класса `TCPConnection`, что дает ему привилегированный доступ к этой операции:

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
  t->ChangeState(s);
}
```

В подклассах `TCPState` реализовано поведение, зависящее от состояния. Соединение TCP может находиться во многих состояниях: `Established` (установлено), `Listening` (прослушивание), `Closed` (закрыто) и т. д., и для каждого из них есть свой подкласс `TCPState`. Мы подробно рассмотрим три подкласса: `TCPEstablished`, `TCPListen` и `TCPClosed`:

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

В подклассах `TCPState` нет никакого локального состояния, поэтому они могут использоваться совместно, так что потребуется только по одному экземпляру каждого класса. Уникальный экземпляр подкласса `TCPState` создается обращением к статической операции `Instance`[1].

В подклассах `TCPState` реализовано зависящее от состояния поведение для тех запросов, которые допустимы в этом состоянии:

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // Послать SYN, получить SYN, ACK и т. д.
    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}
```

[1] Таким образом, каждый подкласс TCPState — это одиночка.

```
void TCPEstablished::Close (TCPConnection* t) {
    // Послать FIN, получить ACK для FIN
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit ( TCPConnection* t, TCPOctetStream* o ) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // Послать SYN, получить SYN, ACK и т. д.
    ChangeState(t, TCPEstablished::Instance());
}
```

После выполнения действий, специфичных для своего состояния, эти операции вызывают `ChangeState` для изменения состояния объекта `TCPConnection`. У него нет никакой информации о протоколе TCP. Именно подклассы `TCPState` определяют переходы между состояниями и действия, диктуемые протоколом.

Известные применения

Ральф Джонсон и Джонатан Цвейг [JZ91] характеризуют паттерн состояние и описывают его применительно к протоколу TCP.

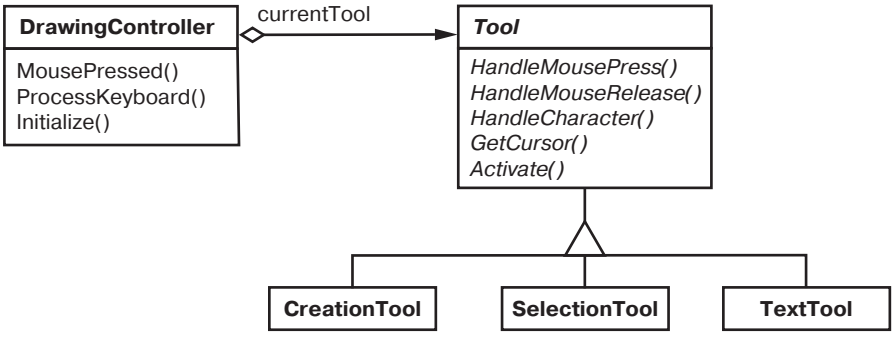
Наиболее популярные интерактивные программы рисования предоставляют «инструменты» для наглядного выполнения операций на экране. Например, инструмент для рисования линий позволяет пользователю щелкнуть в произвольной точке мышью, а затем, перемещая мышь, провести из этой точки линию. Инструмент выбора позволяет выбирать некоторые фигуры. Обычно все имеющиеся инструменты размещаются в палитре. Задача пользователя заключается в том, чтобы правильно выбрать и применить инструмент, но на самом деле поведение редактора изменяется при смене инструмента: при помощи инструмента для рисования мы создаем фигуры, при помощи инструмента выбора — выбираем их и т. д.

Чтобы отразить зависимость поведения редактора от текущего инструмента, можно воспользоваться паттерном состояние.

Можно определить абстрактный класс `Tool`, подклассы которого реализуют поведение, зависящее от инструмента. Графический редактор хранит ссылку на текущий объект Tool и делегирует ему поступающие запросы. При выборе инструмента редактор использует другой объект, что приводит к изменению поведения.

Этот прием используется в каркасах графических редакторов HotDraw [Joh92] и Unidraw [VL90]. Он позволяет клиентам легко определять новые

виды инструментов. В HotDraw класс `DrawingController` переадресует запросы текущему объекту `Tool`. В Unidraw соответствующие классы называются `Viewer` и `Tool`. На приведенной ниже схеме классов схематично представлены интерфейсы классов `Tool` и `DrawingController`.



Описанная Джеймсом Коплиеном [Cop92] идиома «конверт — письмо» (Envelope-Letter) также имеет отношение к паттерну состояние. По сути она представляет собой механизм изменения класса объекта во время выполнения. Паттерн состояние более конкретен; в нем акцент делается на работу с объектами, поведение которых зависит от состояния.

Родственные паттерны

Паттерн приспособленец (231) объясняет, как и когда можно совместно использовать объекты состояния.

Объекты состояния часто бывают одиночками (157).

ПАТТЕРН STRATEGY (СТРАТЕГИЯ)

Название и классификация паттерна

Стратегия — паттерн поведения объектов.

Назначение

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Другие названия

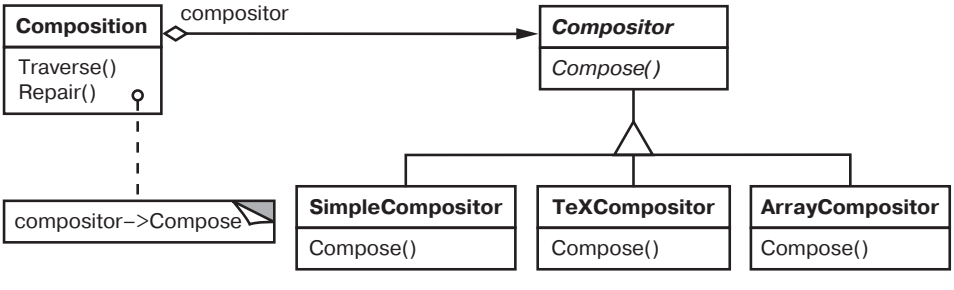
Policy (политика).

Мотивация

Существует много алгоритмов для разбиения текста на строки. Жестко «зашивать» все подобные алгоритмы в классы, которые в них нуждаются, нежелательно по нескольким причинам:

- клиент, которому требуется алгоритм разбиения на строки, усложняется при включении в него соответствующего кода. Таким образом, клиенты становятся более громоздкими и создают больше сложностей в сопровождении, особенно если нужно поддерживать сразу несколько алгоритмов;
- в зависимости от обстоятельств могут применяться разные алгоритмы. Было бы неэффективно поддерживать несколько алгоритмов разбиения на строки, если мы не будем ими пользоваться;
- если разбиение на строки является неотъемлемой частью клиента, то задачи добавления новых и модификации существующих алгоритмов усложняются.

Всех этих проблем можно избежать, если определить классы, инкапсулирующие различные алгоритмы разбиения на строки. Инкапсулированный таким образом алгоритм называется *стратегией*.



Допустим, класс `Composition` отвечает за разбиение на строки текста, отображаемого в окне программы просмотра, и его своевременное обновление. Стратегии разбиения на строки определяются не в классе `Composition`, а в подклассах абстрактного класса `Compositor`. Несколько примеров:

- `SimpleCompositor` реализует простую стратегию, выделяющую по одной строке за раз;

- `TeXCompositor` реализует алгоритм поиска точек разбиения на строки, принятый в редакторе TeX. Эта стратегия пытается оптимизировать разбиение на строки глобально, то есть в целом абзаце;
- `ArrayCompositor` реализует стратегию расстановки переходов на новую строку таким образом, что в каждой строке оказывается одно и то же число элементов. Например, это может быть полезно при построчном отображении набора пиктограмм.

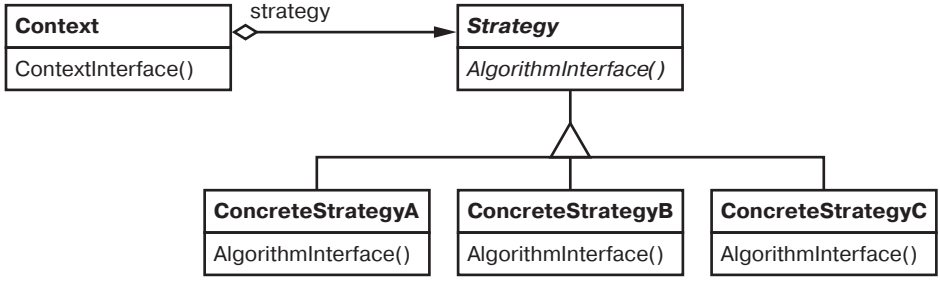
Объект `Composition` хранит ссылку на объект `Compositor`. Всякий раз, когда объекту `Composition` требуется переформатировать текст, он делегирует данную обязанность своему объекту `Compositor`. Чтобы указать, какой объект `Compositor` должен использоваться, клиент встраивает его в объект `Composition`.

Применимость

Основные условия для применения паттерна стратегия:

- *наличие множества родственных классов, отличающихся только поведением*. Стратегия позволяет настроить класс одним из многих возможных вариантов поведения;
- *наличие нескольких разновидностей алгоритма*. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов [HO87];
- в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
- *в классе определено много вариантов поведения, представленных разветвленными условными операторами*. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Структура



Участники

- **Strategy** (`Compositor`) — стратегия:
  - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс `Context` пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе `ConcreteStrategy`;
- **ConcreteStrategy** (`SimpleCompositor`, `TeXCompositor`, `ArrayCompositor`) — конкретная стратегия:
  - реализует алгоритм, использующий интерфейс, объявленный в классе `Strategy`;
- **Context** (`Composition`) — контекст:
  - настраивается объектом класса `ConcreteStrategy`;
  - хранит ссылку на объект класса `Strategy`;
  - может определять интерфейс, который позволяет объекту `Strategy` обращаться к данным контекста.

Отношения

- Классы `Strategy` и `Context` взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые алгоритму данные в момент его вызова. Вместо этого контекст может позволить обращаться к своим операциям в нужные моменты, передавая ссылку на самого себя операциям класса `Strategy`;
- контекст переадресует запросы своих клиентов объекту-стратегии. Обычно клиент создает объект `ConcreteStrategy` и передает его контексту, после чего клиент взаимодействует исключительно с контекстом. Часто в распоряжении клиента находится несколько классов `ConcreteStrategy`, которые он может выбирать.

Результаты

Основные достоинства и недостатки паттерна стратегия:

- *семейства родственных алгоритмов*. Иерархия классов `Strategy` определяет семейство алгоритмов или вариантов поведения, которые можно повторно использовать в разных контекстах. Наследование позволяет вычленить общую для всех алгоритмов функциональность;
- *альтернатива порождению подклассов*. Наследование поддерживает многообразие алгоритмов или поведений. Можно напрямую породить от `Context` подклассы с различными поведениями. Но при этом поведение

жестко «зашивается» в класс `Context`. Реализации алгоритма и контекста смешиваются, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. В результате вы получаете множество родственных классов, отличающихся только алгоритмом или поведением. Инкапсуляция алгоритма в отдельный класс `Strategy` позволяет изменять его независимо от контекста;

- *стратегии позволяют избавиться от условных конструкций*. С паттерном стратегия удается отказаться от условных операторов при выборе нужного поведения. Когда различные поведения помещаются в один класс, трудно выбрать нужное без применения условных операторов. Инкапсуляция же каждого поведения в отдельный класс `Strategy` решает эту проблему.

  Так, без использования стратегий код для разбиения текста на строки мог бы выглядеть следующим образом:

```
void Composition::Repair () {
    switch (_breakingStrategy) {
    case SimpleStrategy:
        ComposeWithSimpleCompositor();
        break;
    case TeXStrategy:
        ComposeWithTeXCompositor();
        break;
        // ...
    }
    // При необходимости объединить результаты
    // с существующей композицией
}
```

  Паттерн стратегия позволяет обойтись без конструкции выбора за счет делегирования задачи разбиения на строки объекту `Strategy`:

```
void Composition::Repair () {
    _compositor->Compose();
    // При необходимости объединить результаты
    // с существующей композицией
}
```

  Если код содержит много условных операторов, то часто это признак того, что нужно применить паттерн стратегия;

- *выбор реализации*. Стратегии могут предлагать различные реализации одного и того же поведения. Клиент вправе выбирать подходящую стратегию в зависимости от своих требований к быстродействию и памяти;

- *клиенты должны знать о различных стратегиях.* Потенциальный недостаток этого паттерна в том, что для выбора подходящей стратегии клиент должен понимать, чем отличаются разные стратегии. Поэтому наверняка придется раскрыть клиенту некоторые особенности реализации. Отсюда следует, что паттерн стратегия стоит применять лишь тогда, когда различия в поведении важны для клиента;
- *затраты на передачу информации между стратегией и контекстом.* Интерфейс `Strategy` совместно используется всеми подклассами `ConcreteStrategy` — какой бы сложной или тривиальной ни была их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые. Это означает, что в отдельных случаях контекст создаст и проинициализирует параметры, которые никому не нужны. Если возникнет проблема, то между классами `Strategy` и `Context` придется установить более тесную связь;
- *увеличение числа объектов.* Применение стратегий увеличивает число объектов в приложении. Иногда эти издержки можно сократить, если реализовать стратегии в виде объектов без состояния, которые могут совместно использоваться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Совместно используемые стратегии не должны сохранять состояние между вызовами. В описании паттерна приспособленец (231) этот подход обсуждается более подробно.

Реализация

Рассмотрим следующие вопросы реализации:

- *определение интерфейсов классов Strategy и Context.* Интерфейсы классов `Strategy` и `Context` должны предоставить объекту класса `ConcreteStrategy` эффективный доступ к любым данным контекста, и наоборот.

  Например, `Context` может передавать данные в параметрах операций класса `Strategy`. Тем самым разрывается тесная связь между контекстом и стратегией. С другой стороны, при этом контекст может передавать данные, которые стратегии не нужны.

  Другой способ — передача *самого контекста* в аргументе. В таком случае стратегия может явно запрашивать у него данные; также стратегия может хранить ссылку на свой контекст, так что передавать вообще ничего не придется. И в том, и в другом случаях стратегия может запрашивать толь-

ко ту информацию, которая реально необходима. Но тогда в контексте должен быть определен более развитый интерфейс к своим данным, что несколько усиливает связанность классов `Strategy` и `Context`.

Выбор подхода зависит от конкретного алгоритма и требований, которые он предъявляет к данным;

- *стратегии как параметры шаблона*. В C++ для настройки класса стратегией можно использовать шаблоны. Этот способ хорош, только если: (1) стратегия определяется на этапе компиляции, и (2) ее не нужно менять во время выполнения. Тогда настраиваемый класс (например, `Context`) определяется в виде шаблона, для которого класс `Strategy` является параметром:

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

Затем этот класс настраивается классом `Strategy` в момент создания экземпляра:

```
class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

При использовании шаблонов отпадает необходимость в абстрактном классе для определения интерфейса `Strategy`. Кроме того, передача стратегии в параметре шаблона позволяет статически связать стратегию с контекстом, вследствие чего повышается эффективность программы;

- *объекты-стратегии можно не задавать*. Класс `Context` можно упростить, если для него нормально не иметь *никакой* стратегии. Прежде чем обращаться к объекту `Strategy`, объект `Context` проверяет наличие стратегии. Если да, то работа продолжается как обычно, в противном случае контекст реализует некое поведение по умолчанию. Преимущество такого подхода в том, что клиентам вообще не нужно иметь дело со стратегиями, *если* их устраивает поведение по умолчанию.

Пример кода

Мы приведем высокоуровневый код для примера из раздела «Мотивация», в основе которого лежат классы `Composition` и `Compositor` из библиотеки InterViews [LCI+92].

В классе `Composition` есть коллекция экземпляров класса `Component`, представляющих текстовые и графические элементы документа. Компоновщик, то есть некоторый подкласс класса `Compositor`, составляет из объектов-компонентов строки, реализуя ту или иную стратегию разбиения на строки. С каждым объектом ассоциирован его естественный размер, а также свойства растягиваемости и сжимаемости. Растягиваемость определяет, насколько возможно увеличивать объект по сравнению с его естественным размером, а сжимаемость — насколько возможно этот размер уменьшать. Композиция передает эти значения компоновщику, который использует их, чтобы найти оптимальное место для разбиения строки.

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;      // Список компонентов
    int _componentCount;         // Количество компонентов
    int _lineWidth;              // Ширина строки в композиции
    int* _lineBreaks;            // Позиции точек разбиения строки
                                 // (измеренные в компонентах)
    int _lineCount;              // Количество строк
};
```

Когда возникает необходимость изменить расположение элементов, композиция запрашивает у компоновщика позиции точек разбиения строк. При этом она передает компоновщику три массива, в которых содержатся естественные размеры, величины растягиваемости и сжимаемости компонентов. Кроме того, передается число компонентов, ширина строки и массив, в который компоновщик должен поместить позиции точек разрыва. Компоновщик возвращает число рассчитанных им точек разбиения.

Интерфейс класса `Compositor` позволяет композиции передать компоновщику всю необходимую ему информацию. Пример передачи данных стратегии:

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
```

```
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

Заметим, что `Compositor` — это абстрактный класс. В его конкретных подклассах определяются различные стратегии разбиения на строки.

Композиция вызывает своего компоновщика из операции `Repair`, которая прежде всего инициализирует массивы, содержащие естественные размеры, растягиваемость и сжимаемость каждого компонента (подробности мы опускаем). Затем `Repair` вызывает компоновщика для получения позиций точек разбиения и, наконец, отображает документ (этот код также опущен):

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // Подготовить массивы с желательными размерами компонентов
    // ...

    // Определить, где должны находиться точки разбиения:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // Разместить компоненты с учетом точек разбиения
    // ...
}
```

Обратимся к подклассам класса `Compositor`. Класс `SimpleCompositor` для определения позиций точек разрыва анализирует компоненты по одному:

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Класс `TeXCompositor` использует более глобальную стратегию. Он рассматривает *абзац* целиком, принимая во внимание размеры и растягиваемость компонентов. Данный класс также пытается минимизировать ширину пропусков между компонентами:

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Класс `ArrayCompositor` разбивает компоненты на строки, оставляя между ними равные промежутки:

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

Не все из этих классов используют в полном объеме информацию, переданную `Compose`. `SimpleCompositor` игнорирует растягиваемость компонентов, принимая во внимание только их естественную ширину. `TeXCompositor` использует всю переданную информацию, а `ArrayCompositor` игнорирует ее.

При создании экземпляра класса `Composition` ему передается компоновщик, которым собираетесь пользоваться:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

Интерфейс класса `Compositor` тщательно спроектирован для поддержки всех алгоритмов размещения, которые могут быть реализованы в подклассах. Вряд ли вам захочется изменять данный интерфейс при появлении каждого нового подкласса, поскольку это означало бы переписывание уже существующих подклассов. В общем случае именно интерфейсы классов `Strategy`

и `Context` определяют, насколько хорошо паттерн `стратегия` соответствует своему назначению.

Известные применения

Библиотеки ET++ [WGM88] и InterViews используют стратегии для инкапсуляции алгоритмов разбиения на строки — так, как мы только что видели.

В системе RTL для оптимизации кода компиляторов [JML92] с помощью стратегий определяются различные схемы распределения регистров (`RegisterAllocator`) и политики управления потоком команд (`RISCscheduler`, `CISCscheduler`). Это позволяет гибко настраивать оптимизатор для разных целевых машинных архитектур.

Каркас ET++ `SwapsManager` предназначен для построения программ, рассчитывающих цены для различных финансовых инструментов [EG92]. Ключевыми абстракциями для него являются `Instrument` (инструмент) и `YieldCurve` (кривая дохода). Различные инструменты реализованы как подклассы класса `Instrument`. `YieldCurve` рассчитывает коэффициенты дисконтирования, на основе которых вычисляется текущее значение будущего движения ликвидности. Оба класса делегируют часть своего поведения объектам-стратегиям класса `Strategy`. В каркасе присутствует семейство конкретных стратегий для генерирования движения ликвидности, оценки оборотов и вычисления коэффициентов дисконтирования. Можно создавать новые механизмы расчетов, конфигурируя классы `Instrument` и `YieldCurve` другими объектами конкретных стратегий. Этот подход поддерживает как использование существующих реализаций стратегий в различных сочетаниях, так и определение новых.

В библиотеке компонентов Гради Буча [BV90] стратегии используются как аргументы шаблонов. В классах коллекций поддерживаются три разновидности стратегий распределения памяти: управляемая (распределение из пула), контролируемая (распределение и освобождение защищаются блокировками) и неуправляемая (стандартное распределение памяти). Стратегия передается классу коллекции в аргументе шаблона в момент создания экземпляра. Например, для коллекции `UnboundedCollection`, в которой используется неуправляемая стратегия, экземпляр создается конструкцией `UnboundedCollection<MyItemType*, Unmanaged>`.

RApp — система для проектирования топологии интегральных схем [GA89, AG90]. Задача RApp — проложить контакты между различными подсистемами на схеме. Алгоритмы трассировки в RApp определены как подклассы абстрактного класса `Router`, который является стратегией.

В библиотеке ObjectWindows фирмы Borland [Bor94] стратегии используются в диалоговых окнах для проверки правильности введенных пользователем данных. Например, можно контролировать, что число принадлежит заданному диапазону, а в данном поле должны быть только цифры. Не исключено, что проверка корректности введенной строки потребует поиска данных по справочной таблице.

Для инкапсуляции стратегий проверки в ObjectWindows используются объекты класса `Validator` — частный случай паттерна стратегия. Поля для ввода данных делегируют стратегию контроля необязательному объекту `Validator`. Клиент при необходимости присоединяет таких проверяющих к полю (пример необязательной стратегии). В момент закрытия диалогового окна поля «просят» своих контролеров проверить правильность данных. В библиотеке имеются классы контролеров для наиболее распространенных случаев, например `RangeValidator` для проверки принадлежности числа диапазону. Но клиент может легко определить и собственные стратегии проверки, порождая подклассы от класса `Validator`.

Родственные паттерны

Приспособленец (231): объекты-стратегии в большинстве случаев подходят для применения паттерна приспособленец.

ПАТТЕРН TEMPLATE METHOD (ШАБЛОННЫЙ МЕТОД)

Название и классификация паттерна

Шаблонный метод — паттерн поведения классов.

Назначение

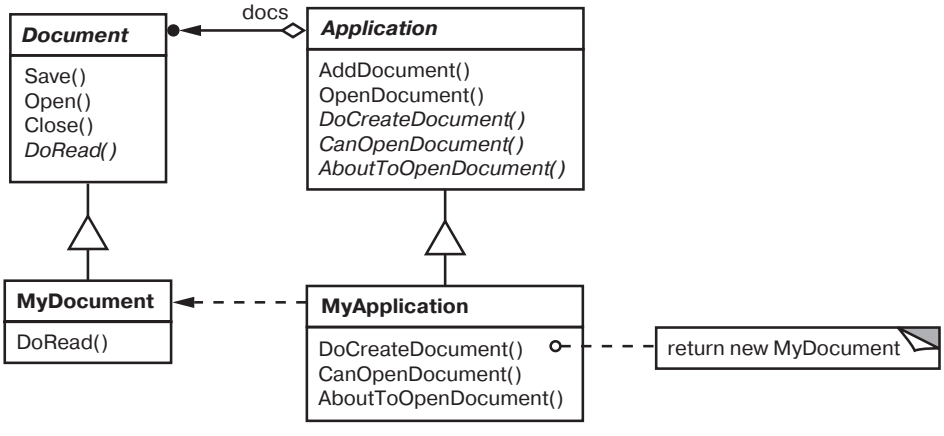
Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

Мотивация

Рассмотрим каркас приложения, в котором имеются классы `Application` и `Document`. Класс `Application` отвечает за открытие существующих документов, хранящихся во внешнем формате (например, в файле). Объект класса `Document` представляет информацию документа после его прочтения из файла.

Приложения, построенные на базе этого каркаса, могут порождать подклассы от классов `Application` и `Document`, отвечающие конкретным потребностям.

Например, графический редактор определит подклассы `DrawApplication` и `DrawDocument`, а электронная таблица — подклассы `SpreadsheetApplication` и `SpreadsheetDocument`.



В абстрактном классе `Application` определен алгоритм открытия и чтения документа в операции `OpenDocument`:

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // Обработать документ невозможно
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

Операция `OpenDocument` определяет все шаги открытия документа. Она проверяет, возможно ли открыть документ, создает объект класса `Document`, добавляет его к набору документов и читает документ из файла.

Операцию вида `OpenDocument` мы будем называть *шаблонным методом*, описывающим алгоритм в категориях абстрактных операций, которые замещены в подклассах для получения нужного поведения. Подклассы класса

`Application` проверяют возможность открытия (`CanOpenDocument`) и создания документа (`DoCreateDocument`). Подклассы класса `Document` считывают документ (`DoRead`). Шаблонный метод определяет также операцию, которая позволяет подклассам `Application` получить информацию о том, что документ вот-вот будет открыт (`AboutToOpenDocument`).

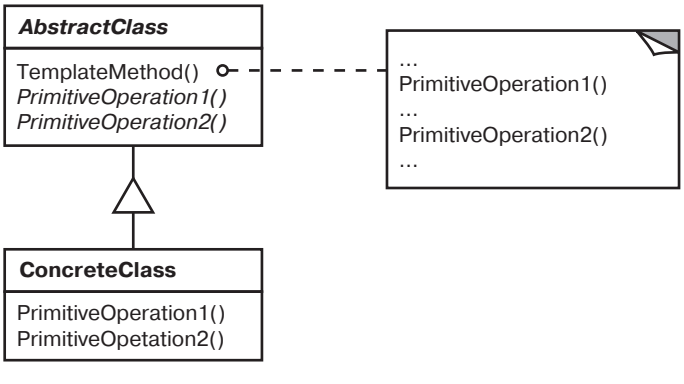
Определяя некоторые шаги алгоритма с помощью абстрактных операций, шаблонный метод фиксирует их последовательность, но позволяет реализовать их в подклассах классов `Application` и `Document`.

Применимость

Основные условия для применения паттерна шаблонный метод:

- однократное использование инвариантных частей алгоритма, при этом реализация изменяющегося поведения остается на усмотрение подклассов;
- необходимость вычленить и локализовать в одном классе поведение, общее для всех подклассов, чтобы избежать дублирования кода. Это хороший пример техники «вынесения за скобки с целью обобщения», описанной в работе Уильяма Опдайка (William Opdyke) и Ральфа Джонсона (Ralph Johnson) [OJ93]. Сначала выявляются различия в существующем коде, которые затем выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;
- управление расширениями подклассов. Шаблонный метод можно определить так, что он будет вызывать операции-зацепки (hooks) — см. раздел «Результаты» — в определенных точках, разрешив тем самым расширение только в этих точках.

Структура



Участники

- **AbstractClass** (`Application`) — абстрактный класс:
  - определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма;
  - реализует шаблонный метод, определяющий скелет алгоритма. Шаблонный метод вызывает примитивные операции, а также операции, определенные в классе AbstractClass или в других объектах;
- **ConcreteClass** (`MyApplication`) — конкретный класс:
  - реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.

Отношения

`ConcreteClass` предполагает, что инвариантные шаги алгоритма будут выполняться в `AbstractClass`.

Результаты

Шаблонные методы — один из фундаментальных приемов повторного использования кода. Они играют особенно важную роль в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы.

Шаблонные методы приводят к инвертированной структуре кода, которую иногда называют принципом Голливуда, подразумевая часто употребляемую в этой киноимперии фразу «Не звоните нам, мы сами вам позвоним» [Swe85]. В данном случае это означает, что родительский класс вызывает операции подкласса, а не наоборот.

Шаблонные методы вызывают операции следующих видов:

- конкретные операции (либо из класса `ConcreteClass`, либо из классов клиента);
- конкретные операции из класса `AbstractClass` (то есть операции, полезные всем подклассам);
- примитивные операции (то есть абстрактные операции);
- фабричные методы (см. паттерн фабричный метод (135));
- операции-зацепки (hook operations), реализующие поведение по умолчанию, которое может быть расширено в подклассах. Часто такая операция по умолчанию не делает ничего.

Важно, чтобы в шаблонном методе четко различались операции-зацепки (которые *можно* замещать) и абстрактные операции (которые *нужно* замещать). Чтобы повторно использовать абстрактный класс с максимальной эффективностью, авторы подклассов должны понимать, какие операции предназначены для замещения.

Подкласс может расширить поведение некоторой операции, заместив ее и явно вызвав эту операцию из родительского класса:

```
void DerivedClass::Operation () {
    // Расширенное поведение DerivedClass
    ParentClass::Operation();
}
```

К сожалению, очень легко забыть о необходимости вызывать унаследованную операцию. Такую операцию можно трансформировать в шаблонный метод, чтобы предоставить родителю контроль над тем, как подклассы расширяют его. Идея в том, чтобы вызывать операцию-зацепку из шаблонного метода в родительском классе. Тогда подклассы смогут переопределить именно эту операцию:

```
void ParentClass::Operation () {
    // Поведение ParentClass
    HookOperation();
}
```

В родительском классе `ParentClass` операция `HookOperation` не делает ничего:

```
void ParentClass::HookOperation () { }
```

Подклассы переопределяют `HookOperation`, чтобы расширить свое поведение:

```
void DerivedClass::HookOperation () {
  // Расширение в производном классе
}
```

Реализация

При реализации паттерна шаблонный метод следует обратить внимание на следующие аспекты:

- *использование контроля доступа в C++*. В этом языке примитивные операции, которые вызывает шаблонный метод, можно объявить защищенными членами. Тогда гарантируется, что вызывать их сможет только сам шаблонный метод. Примитивные операции, которые *обязательно* нужно замещать, объявляются как чисто виртуальные функции. Сам

шаблонный метод замещать не надо, так что его можно сделать невиртуальной функцией-членом;

- *сокращение числа примитивных операций*. Важной целью при проектировании шаблонных методов является всемерное сокращение числа примитивных операций, которые должны быть замещены в подклассах. Чем больше операций нужно замещать, тем утомительнее становится программирование клиента;
- *соглашение об именах*. Выделить операции, которые необходимо заместить, можно путем добавления к их именам некоторого префикса. Например, в каркасе MacApp для приложений на платформе Macintosh [App89] имена шаблонных методов начинаются с префикса `Do`: `DoCreateDocument`, `DoRead` и т. д.

Пример кода

Следующий пример на языке C++ показывает, как родительский класс может навязать своим подклассам некоторый инвариант. Пример взят из библиотеки NeXT AppKit [Add94]. Рассмотрим класс `View`, поддерживающий рисование на экране, — своего рода инвариант, который заключается в том, что подклассы могут изменять вид только тогда, когда он находится в фокусе. Для этого необходимо, чтобы был установлен определенный контекст рисования (например, цвета и шрифты).

Для установки состояния можно воспользоваться шаблонным методом `Display`. В классе `View` определены две конкретные операции (`SetFocus` и `ResetFocus`), которые соответственно устанавливают и сбрасывают контекст рисования. Операция-зацепка `DoDisplay` класса `View` занимается собственно рисованием. `Display` вызывает `SetFocus` перед `DoDisplay`, чтобы подготовить контекст, и `ResetFocus` после `DoDisplay` — чтобы его сбросить:

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

С целью поддержки инварианта клиенты класса `View` всегда вызывают `Display` и подклассы `View` всегда замещают `DoDisplay`.

В классе `View` операция `DoDisplay` не делает ничего:

```
void View::DoDisplay () { }
```

Подклассы переопределяют ее, чтобы добавить свое конкретное поведение рисования:

```
void MyView::DoDisplay () {
    // изобразить содержимое вида
}
```

Известные применения

Шаблонные методы настолько фундаментальны, что встречаются почти в каждом абстрактном классе. В работах Ребекки Вирфс-Брок и др. [WBWW90, WBJ90] подробно обсуждаются шаблонные методы.

Родственные паттерны

Фабричные методы (135) часто вызываются из шаблонных. В примере из раздела «Мотивация» шаблонный метод `OpenDocument` вызывал фабричный метод `DoCreateDocument`.

Стратегия (362): шаблонные методы применяют наследование для модификации части алгоритма. Стратегии используют делегирование для модификации алгоритма в целом.

ПАТТЕРН VISITOR (ПОСЕТИТЕЛЬ)

Название и классификация паттерна

Посетитель — паттерн поведения объектов.

Назначение

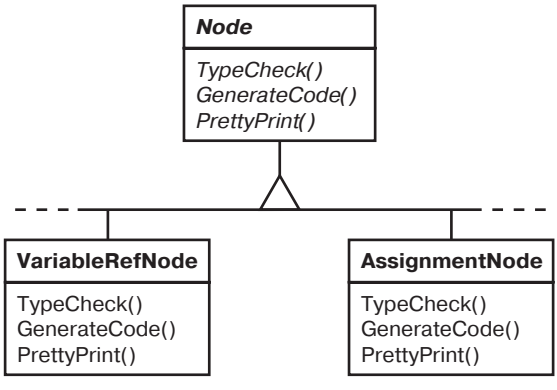
Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

Мотивация

Рассмотрим компилятор, который представляет программу в виде абстрактного синтаксического дерева. Над такими деревьями он должен выполнять операции «статического семантического» анализа, например проверять, что все переменные определены. Еще ему нужно генерировать код. Аналогично можно было бы определить операции контроля типов, оптимизации кода,

анализа потока выполнения, проверки того, что каждой переменной было присвоено конкретное значение перед первым использованием, и т. д. Более того, абстрактные синтаксические деревья могли бы служить для красивого оформления результатов программы, реструктурирования кода и вычисления различных метрик программы.

В большинстве таких операций узлы дерева, представляющие операторы присваивания, должны рассматриваться иначе, чем узлы, представляющие переменные и арифметические выражения. Поэтому один класс будет создан для операторов присваивания, другой — для доступа к переменным, третий — для арифметических выражений и т. д. Набор классов узлов, конечно, зависит от компилируемого языка, но не очень сильно.



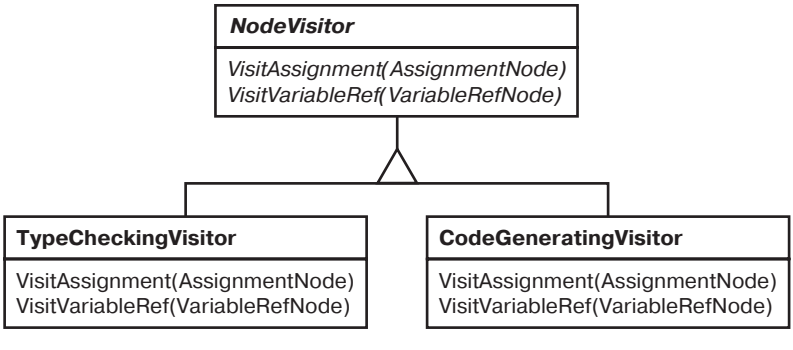
На схеме показана часть иерархии классов `Node`. Проблема здесь в том, что при распределении всех операций по классам различных узлов получится система, которую трудно понять, сопровождать и изменять. Вряд ли кто-нибудь разберется в программе, если код, отвечающий за проверку типов, будет перемешан с кодом, реализующим красивую печать или анализ потока выполнения. Кроме того, добавление любой новой операции потребует перекомпиляции всех классов. Оптимальный вариант — наличие возможности добавлять операции по отдельности и отсутствие зависимости классов узлов от применяемых к ним операций.

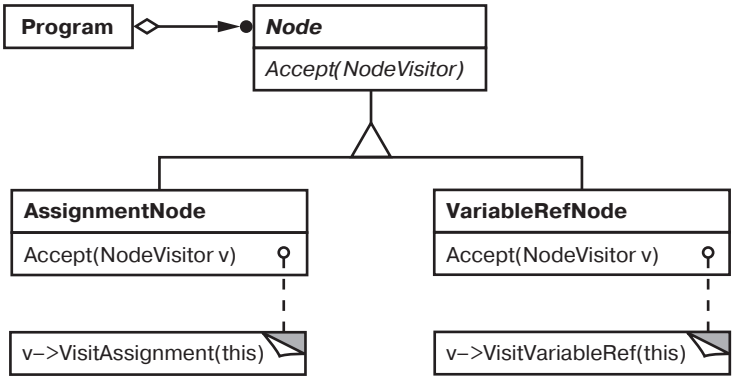
Обе проблемы можно решить выделением взаимосвязанных операций из каждого класса в отдельный объект, называемый *посетителем*, и передавать его элементам абстрактного синтаксического дерева по мере обхода. «При-

нимая» посетителя, элемент посылает ему запрос, в котором содержится, в частности, класс элемента. Кроме того, в запросе присутствует в виде аргумента и сам элемент. Посетителю в данной ситуации предстоит выполнить операцию над элементом — ту самую, которая наверняка находилась бы в классе элемента.

Например, компилятор, который не использует посетителей, мог бы проверить тип процедуры, вызвав операцию `TypeCheck` для представляющего ее абстрактного синтаксического дерева. Каждый узел дерева должен был реализовать операцию `TypeCheck` путем рекурсивного вызова ее же для своих компонентов (см. приведенную выше схему классов). Если же компилятор проверяет тип процедуры посредством посетителей, то ему достаточно создать объект класса `TypeCheckingVisitor` и вызвать для дерева операцию `Accept`, передав ей этот объект в аргументе. Каждый узел должен был реализовать `Accept` путем обращения к посетителю: узел, соответствующий оператору присваивания, вызывает операцию посетителя `VisitAssignment`, а узел, ссылающийся на переменную, — операцию `VisitVariableReference`. То, что раньше было операцией `TypeCheck` в классе `AssignmentNode`, стало операцией `VisitAssignment` в классе `TypeCheckingVisitor`.

Чтобы посетители могли заниматься не только проверкой типов, нам понадобится абстрактный класс `NodeVisitor`, являющийся родителем для всех посетителей синтаксического дерева. Приложение, которому нужно вычислять метрики программы, определило бы новые подклассы `NodeVisitor`, так что нам не пришлось бы добавлять зависящий от приложения код в классы узлов. Паттерн посетитель инкапсулирует операции, выполняемые на каждой фазе компиляции, в классе `Visitor`, ассоциированном с этой фазой.





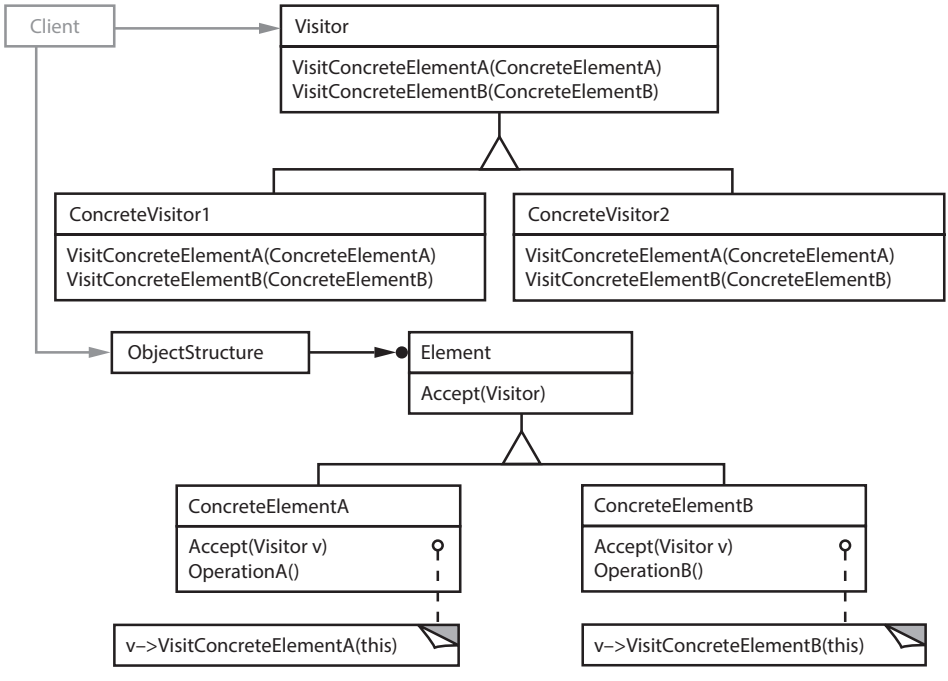
Применяя паттерн посетитель, вы определяете две иерархии классов: одну для элементов, над которыми выполняется операция (иерархия `Node`), а другую — для посетителей, описывающих те операции, которые выполняются над элементами (иерархия `NodeVisitor`). Новая операция создается путем добавления подкласса в иерархию классов посетителей. До тех пор пока грамматика языка остается постоянной (то есть не добавляются новые подклассы `Node`), новую функциональность можно получить путем определения новых подклассов `NodeVisitor`.

Применимость

Основные условия для применения паттерна посетитель:

- в структуре присутствуют объекты многих классов с различными интерфейсами, и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- над объектами, входящими в состав структуры, должны выполняться разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции;
- классы, определяющие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, придется переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Структура



Участники

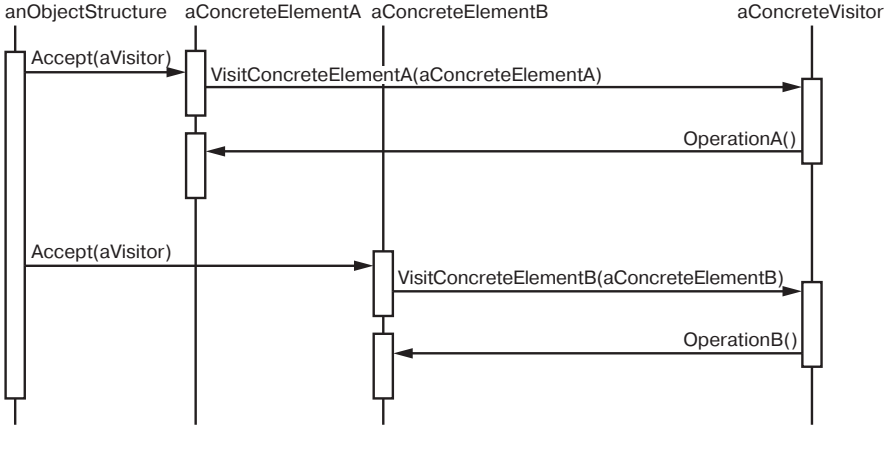
- **Visitor** (`NodeVisitor`) — посетитель:
  - объявляет операцию `Visit` для каждого класса `ConcreteElement` в структуре объектов. Имя и сигнатура этой операции идентифицируют класс, который посылает посетителю запрос `Visit`. Это позволяет посетителю определить, элемент какого конкретного класса он посещает. Владея такой информацией, посетитель может обращаться к элементу напрямую через его интерфейс;
- **ConcreteVisitor** (`TypeCheckingVisitor`) — конкретный посетитель:
  - реализует все операции, объявленные в классе `Visitor`. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс `ConcreteVisitor` предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры;

- **Element** (`Node`) — элемент:
  - определяет операцию `Accept`, которая принимает посетителя в аргументе;
- **ConcreteElement** (`AssignmentNode`, `VariableRefNode`) — конкретный элемент:
  - реализует операцию `Accept`, принимающую посетителя как аргумент;
- **ObjectStructure** (`Program`) — структура объектов:
  - может перечислить свои элементы;
  - может предоставить посетителю высокоуровневый интерфейс для посещения своих элементов;
  - может быть как составным объектом (см. паттерн компоновщик (196)), так и коллекцией, например списком или множеством.

Отношения

- Клиент, использующий паттерн посетитель, должен создать объект класса `ConcreteVisitor`, а затем обойти всю структуру, посетив каждый ее элемент.
- При посещении элемента последний вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в аргументе, чтобы посетитель мог при необходимости получить доступ к его состоянию.

  На представленной диаграмме взаимодействий показаны отношения между объектом, структурой, посетителем и двумя элементами.



Результаты

Основные достоинства и недостатки паттерна посетитель:

- *упрощение добавления новых операций*. С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя. Напротив, если функциональность распределена по нескольким классам, то для определения новой операции придется изменить каждый класс;
- *объединение родственных операций и отсечение тех, которые не имеют к ним отношения*. Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе. Не связанные друг с другом функции распределяются по отдельным подклассам класса `Visitor`. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в посетителях. Все относящиеся к алгоритму структуры данных можно скрыть в посетителе;
- *трудности с добавлением новых классов ConcreteElement*. Паттерн посетитель усложняет добавление новых подклассов класса `Element`. Каждый новый конкретный элемент требует объявления новой абстрактной операции в классе `Visitor`, которую нужно реализовать в каждом из существующих классов `ConcreteVisitor`. Иногда большинство конкретных посетителей могут унаследовать операцию по умолчанию, предоставляемую классом `Visitor`, что скорее исключение, чем правило. Поэтому при решении вопроса о том, стоит ли использовать паттерн посетитель, нужно прежде всего посмотреть, что будет изменяться чаще: алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Вполне вероятно, что с сопровождением иерархии классов `Visitor` возникнут трудности, если новые классы `ConcreteElement` добавляются часто. В таких случаях проще определить операции прямо в классах, представленных в структуре. Если же иерархия классов `Element` стабильна, но постоянно расширяется набор операций или модифицируются алгоритмы, то паттерн посетитель поможет лучше управлять такими изменениями;
- *посещение различных иерархий классов*. Итератор (см. описание паттерна итератор) может посещать объекты структуры по мере ее обхода, вызывая операции объектов. Но итератор не способен работать со структурами, состоящими из объектов разных типов. Так, интерфейс класса `Iterator`, рассмотренный на с. 310, может всего лишь получить доступ к объектам типа `Item`:

```
template <class Item>
class Iterator {
  // ...
  Item CurrentItem() const;
};
```

Отсюда следует, что все элементы, которые итератор может посетить, должны иметь общий родительский класс `Item`.

У посетителя таких ограничений нет. Ему разрешено посещать объекты, не имеющие общего родительского класса. В интерфейс класса `Visitor` можно добавить операции для объектов любого типа. Например, в следующем объявлении

```
class Visitor {
public:
  // ...
  void VisitMyType(MyType*);
  void VisitYourType(YourType*);
};
```

классы `MyType` и `YourType` необязательно должны быть связаны отношением наследования;

- *накопление состояния*. Посетители могут накапливать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, то состояние придется передавать в дополнительных аргументах операций, выполняющих обход, или хранить в глобальных переменных;
- *нарушение инкапсуляции*. Применение посетителей подразумевает, что класс `ConcreteElement` имеет достаточно развитый интерфейс, для того чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что ставит под угрозу инкапсуляцию.

Реализация

С каждым объектом структуры ассоциируется некий класс посетителя `Visitor`. В этом абстрактном классе объявлены операции `VisitConcreteElement` для каждого конкретного класса `ConcreteElement` элементов, представленных в структуре. В каждой операции типа `Visit` аргумент объявлен как принадлежащий одному из классов ConcreteElement, так что посетитель может напрямую обращаться к интерфейсу этого класса. Классы `ConcreteVisitor`

замещают операции `Visit` с целью реализации поведения посетителя для соответствующего класса `ConcreteElement`.

В C++ объявление класса `Visitor` выглядело бы примерно так:

```
class Visitor {
public:
  virtual void VisitElementA(ElementA*);
  virtual void VisitElementB(ElementB*);

  // И т. д. для других конкретных элементов
protected:
  Visitor();
};
```

Каждый класс `ConcreteElement` реализует операцию `Accept`, которая вызывает соответствующую операцию `Visit...` посетителя для этого класса. Следовательно, вызываемая в конечном итоге операция зависит как от класса элемента, так и от класса посетителя[1].

Объявления конкретных элементов выглядят так:

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};

class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```

[1] Можно было бы использовать перегрузку функций, чтобы дать этим операциям одно и то же простое имя (например, Visit), так как они уже различаются типом передаваемого параметра. Имеются аргументы как в пользу подобной перегрузки, так и против нее. С одной стороны, подчеркивается, что все операции выполняют однотипный анализ, хотя и с разными аргументами. С другой стороны, при этом читателю программы может быть не вполне понятно, что происходит при вызове. В общем все зависит от того, часто ли вы применяете перегрузку функций.

Класс `CompositeElement` мог бы реализовать операцию `Accept` следующим образом:

```
class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```

При решении вопроса о применении паттерна посетитель часто возникают два спорных момента:

- *двойная диспетчеризация*. По своей сути паттерн посетитель добавляет в классы новые операции без их изменения. Это делается с помощью приема, называемого двойной диспетчеризацией. Данная техника хорошо известна. Некоторые языки программирования (например, CLOS) поддерживают ее явно. Языки же вроде C++ и Smalltalk поддерживают только одинарную диспетчеризацию.

  Для определения того, какая операция будет выполнять запрос, в языках с одинарной диспетчеризацией необходимы имя запроса и тип получателя. Например, то, какая операция будет вызвана для обработки запроса `GenerateCode`, зависит от типа объекта в узле, которому адресован запрос. В C++ вызов `GenerateCode` для экземпляра `VariableRefNode` приводит к вызову функции `VariableRefNode::GenerateCode` (генерирующей код обращения к переменной). Вызов же `GenerateCode` для узла класса `AssignmentNode` приводит к вызову функции `AssignmentNode::GenerateCode` (генерирующей код для оператора присваивания). Таким образом, выполняемая операция определяется одновременно видом запроса и типом получателя.

  Понятие «двойная диспетчеризация» означает, что выполняемая операция зависит от вида запроса и типов двух получателей. `Accept` — это операция с двойной диспетчеризацией. Ее семантика зависит от типов двух объектов: `Visitor` и `Element`. Двойная диспетчеризация позволяет

посетителю запрашивать разные операции для каждого класса элемента[1].

Поэтому возникает необходимость в паттерне посетитель: выполняемая операция зависит и от типа посетителя, и от типа посещаемого элемента. Вместо статической привязки операций к интерфейсу класса `Element` мы можем консолидировать эти операции в классе `Visitor` и использовать `Accept` для привязки их во время выполнения. Расширение интерфейса класса `Element` сводится к определению нового подкласса `Visitor`, а не к модификации многих подклассов `Element`;

- *какой участник несет ответственность за обход структуры.* Посетитель должен обойти каждый элемент структуры объектов. Вопрос в том, как туда попасть. Ответственность за обход можно возложить на саму структуру объектов, на посетителя или на отдельный объект-итератор (см. паттерн итератор (302)). Чаще всего структура объектов отвечает за обход. Коллекция просто обходит все свои элементы, вызывая для каждого операцию `Accept`. Составной объект обычно обходит самого себя, заставляя операцию `Accept` посетить потомков текущего элемента и рекурсивно вызвать `Accept` для каждого из них.

  Другое решение — воспользоваться итератором для посещения элементов. В C++ можно применить внутренний или внешний итератор, в зависимости от того, что доступно и более эффективно. В Smalltalk обычно работают с внутренним итератором на основе метода `do:` и блока. Поскольку внутренние итераторы реализуются самой структурой объектов, то работа с ними во многом напоминает предыдущее решение, когда за обход отвечает структура. Основное различие заключается в том, что внутренний итератор не приводит к двойной диспетчеризации: он вызывает операцию *посетителя с элементом* в качестве аргумента, а не операцию *элемента с посетителем* в качестве аргумента. Однако использовать паттерн посетитель с внутренним итератором легко в том случае, когда операция посетителя вызывает операцию элемента без рекурсии.

  Можно даже включить алгоритм обхода в посетителя, хотя закончится это дублированием кода обхода в каждом классе `ConcreteVisitor` для

[1] Если есть двойная диспетчеризация, то почему бы не быть тройной, четверной или диспетчеризации произвольной кратности? Двойная диспетчеризация — это просто частный случай множественной диспетчеризации, при которой выбираемая операция зависит от любого числа типов. (CLOS как раз и поддерживает множественную диспетчеризацию.) В языках с поддержкой двойной или множественной диспетчеризации необходимость в паттерне посетитель возникает гораздо реже.

каждого агрегата `ConcreteElement`. Основная причина такого решения — необходимость реализовать особо сложную стратегию обхода, зависящую от результатов операций над объектами структуры. Этот случай рассматривается в разделе «Пример кода».

Пример кода

Поскольку посетители обычно ассоциируются с составными объектами, то для иллюстрации паттерна посетитель мы воспользуемся классами `Equipment`, определенными в разделе «Пример кода» из описания паттерна компоновщик (196). Для определения операций, создающих инвентарную опись материалов и вычисляющих полную стоимость агрегата, нам понадобится паттерн посетитель. Классы `Equipment` настолько просты, что применять паттерн посетитель в общем-то излишне, но на этом примере демонстрируются основные особенности его реализации.

Приведем еще раз объявление класса `Equipment` из описания паттерна компоновщик (196). Мы добавили операцию `Accept`, чтобы можно было работать с посетителем:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Операции класса `Equipment` возвращают такие атрибуты единицы оборудования, как энергопотребление и стоимость. В подклассах эти операции переопределены в соответствии с конкретными типами оборудования (рама, дисководы и электронные платы).

В абстрактном классе всех посетителей оборудования имеются виртуальные функции для каждого подкласса (см. ниже). По умолчанию эти функции ничего не делают:

```
class EquipmentVisitor {
public:
  virtual ~EquipmentVisitor();

  virtual void VisitFloppyDisk(FloppyDisk*);
  virtual void VisitCard(Card*);
  virtual void VisitChassis(Chassis*);
  virtual void VisitBus(Bus*);

  // И так далее для всех конкретных подклассов Equipment
protected:
  EquipmentVisitor();
};
```

Все подклассы класса `Equipment` определяют функцию `Accept` практически одинаково. Она вызывает операцию `EquipmentVisitor`, которая соответствует классу, получившему запрос `Accept`:

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
  visitor.VisitFloppyDisk(this);
}
```

Виды оборудования, которые содержат другое оборудование (в частности, подклассы `CompositeEquipment` в терминологии паттерна компоновщик), реализуют `Accept` путем обхода своих потомков и вызова `Accept` для каждого из них. Затем, как обычно, вызывается операция `Visit`. Например, `Chassis::Accept` могла бы обойти все расположенные на шасси компоненты следующим образом:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
    for (
        ListIterator i(_parts);
        !i.IsDone();
        i.Next()
    ) {
         i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

Подклассы `EquipmentVisitor` определяют конкретные алгоритмы, применяемые к структуре оборудования. Так, `PricingVisitor` вычисляет стоимость всей конструкции, для чего суммирует нетто-цены простых компонентов (например, гибкие диски) и цену со скидкой составных компонентов (например, рамы и шины):

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

Таким образом, посетитель `PricingVisitor` подсчитает полную стоимость всех узлов конструкции. Заметим, что `PricingVisitor` выбирает стратегию вычисления цены в зависимости от класса оборудования, для чего вызывает соответствующую функцию класса. Особенно важно то, что для оценки конструкции можно выбрать другую стратегию, просто поменяв класс `PricingVisitor`.

Определить посетителя для составления инвентарной описи можно следующим образом:

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...

private:
    Inventory _inventory;
};
```

Посетитель `InventoryVisitor` подсчитывает итоговое количество каждого вида оборудования во всей конструкции. При этом используется класс `Inventory`, в котором определен интерфейс для добавления компонента (здесь мы его приводить не будем):

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

Добавление `InventoryVisitor` к структуре объектов может происходить следующим образом:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
    << component->Name()
    << visitor.GetInventory();
```

Далее мы покажем, как на языке Smalltalk реализовать пример из описания паттерна интерпретатор (287) с помощью паттерна посетитель. Как и в предыдущем случае, этот пример настолько мал, что паттерн посетитель особой пользы не принесет, но неплохо демонстрирует основные принципы. Кроме того, демонстрируется ситуация, в которой за обход отвечает посетитель.

Структура объектов (регулярные выражения) представлена четырьмя классами, в каждом из которых существует метод `accept:`, принимающий посетитель в качестве аргумента. В классе `SequenceExpression` метод `accept:` выглядит так:

```
accept: aVisitor
    ^ aVisitor visitSequence: self
```

Метод `accept:` в классах `RepeatExpression`, `AlternationExpression` и `LiteralExpression` посылает сообщения `visitRepeat:`, `visitAlternation:` и `visitLiteral:` соответственно.

Все четыре класса должны иметь функции доступа, к которым может обратиться посетитель. Для `SequenceExpression` это `expression1` и `expression2`;

для `AlternationExpression` — `alternative1` и `alternative2`; для класса `RepeatExpression` — `repetition`, а для `LiteralExpression` — `components`.

Конкретным посетителем выступает класс `REMatchingVisitor`. Он отвечает за обход структуры, поскольку алгоритм обхода нерегулярен. В основном это происходит из-за того, что `RepeatExpression` посещает свой компонент многократно. В классе `REMatchingVisitor` есть переменная экземпляра `inputState`. Его методы практически повторяют методы `match:` классов выражений из паттерна интерпретатор, только вместо аргумента `inputState` подставляется узел, описывающий сравниваемое выражение. Однако они по-прежнему возвращают множество потоков, с которыми должно быть сопоставлено выражение для получения текущего состояния:

```
visitSequence: sequenceExp
    inputState := sequenceExp expression1 accept: self.
    ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
    | finalState |
    finalState := inputState copy.
    [inputState isEmpty]
        whileFalse:
            [inputState := repeatExp repetition accept: self.
            finalState addAll: inputState].
    ^ finalState

visitAlternation: alternateExp
    | finalState originalState |
    originalState := inputState.
    finalState := alternateExp alternative1 accept: self.
    inputState := originalState.
    finalState addAll: (alternateExp alternative2 accept: self).
    ^ finalState

visitLiteral: literalExp
    | finalState tStream |
    finalState := Set new.
    inputState
        do:
            [:stream | tStream := stream copy.
                (tStream nextAvailable:
                    literalExp components size
                ) = literalExp components
                    ifTrue: [finalState add: tStream]
            ].
    ^ finalState
```

Известные применения

В компиляторе Smalltalk-80 имеется класс посетителя, который называется `ProgramNodeEnumerator`. В основном он применяется в алгоритмах анализа исходного текста программы и не используется ни для генерирования кода, ни для красивой печати, хотя мог бы.

IRIS Inventor [Str93] — это библиотека для разработки приложений трехмерной графики. Библиотека представляет собой трехмерную сцену в виде иерархии узлов, каждый из которых соответствует либо геометрическому объекту, либо его атрибуту. Для операций типа изображения сцены или обработки события ввода необходимо по-разному обходить эту иерархию. В Inventor для этого служат посетители, которые называются действиями (actions). Есть различные посетители для изображения, обработки событий, поиска, сохранения и определения ограничивающих прямоугольников.

Чтобы упростить добавление новых узлов, в библиотеке Inventor реализована схема двойной диспетчеризации на C++. Для этого служит информация о типе, доступная во время выполнения, и двумерная таблица, строки которой представляют посетителей, а колонки — классы узлов. В каждой ячейке хранится указатель на функцию, связанную с парой «посетитель — класс» узла.

Марк Линтон (Mark Linton) ввел термин «посетитель» (Visitor) в спецификацию библиотеки для построения приложений X Consortium's Fresco Application Toolkit [LP93].

Родственные паттерны

Компоновщик (196): посетители могут использоваться для выполнения операции над всеми объектами структуры, определенной с помощью паттерна компоновщик.

Интерпретатор (287): посетитель может использоваться для выполнения интерпретации.

ОБСУЖДЕНИЕ ПАТТЕРНОВ ПОВЕДЕНИЯ

ИНКАПСУЛЯЦИЯ ВАРИАЦИЙ

Инкапсуляция вариаций — элемент многих паттернов поведения. Если определенная часть программы подвержена периодическим изменениям, эти паттерны позволяют определить объект для инкапсуляции такого аспекта.

Другие части программы, зависящие от данного аспекта, могут кооперироваться с ним. Обычно паттерны поведения определяют абстрактный класс, с помощью которого описывается инкапсулирующий объект[1]. Своим названием паттерн как раз и обязан этому объекту.

Например:

- объект стратегия (362) инкапсулирует алгоритм;
- объект состояние (352) инкапсулирует поведение, зависящее от состояния;
- объект посредник (319) инкапсулирует протокол общения между объектами;
- объект итератор (302) инкапсулирует способы доступа и обхода компонентов составного объекта.

Перечисленные паттерны описывают подверженные изменениям аспекты программы. В большинстве паттернов фигурируют два вида объектов: новый объект (или объекты), который инкапсулирует аспект, и существующий объект (или объекты), который пользуется новыми. Если бы не паттерн, то функциональность новых объектов пришлось бы делать неотъемлемой частью существующих. Например, код объекта-стратегии, вероятно, был бы «зашит» в контекст стратегии, а код объекта-состояния был бы реализован непосредственно в контексте состояния.

Впрочем, не все паттерны поведения разбивают функциональность таким образом. Например, паттерн цепочка обязанностей (263) связан с произвольным числом объектов (то есть цепочкой), причем все они могут уже существовать в системе.

Цепочка обязанностей иллюстрирует еще одно различие между паттернами поведения: не все они определяют статические отношения взаимосвязи между классами. В частности, цепочка обязанностей показывает, как организовать обмен информацией между заранее неизвестным числом объектов. В других паттернах участвуют объекты, передаваемые в качестве аргументов.

---

[1] Эта тема красной нитью проходит и через другие паттерны. Абстрактная фабрика, строитель и прототип инкапсулируют знание о том, как создаются объекты. Декоратор инкапсулирует обязанности, которые могут быть добавлены к объекту. Мост отделяет абстракцию от ее реализации, позволяя изменять их независимо друг от друга.

ОБЪЕКТЫ КАК АРГУМЕНТЫ

В нескольких паттернах участвует объект, *всегда* используемый только как аргумент. Одним из них является посетитель (379). Объект-посетитель — это аргумент полиморфной операции `Accept`, принадлежащей посещаемому объекту. Посетитель никогда не рассматривается как часть посещаемых объектов, хотя традиционным альтернативным вариантом этому паттерну служит распределение кода посетителя между классами объектов, входящих в структуру.

Другие паттерны определяют объекты, выступающие в роли волшебных сущностей, которые передаются от одного владельца к другому и активизируются в будущем. К этой категории относятся команда (275) и хранитель (330). В паттерне команда такой «палочкой» является запрос, а в хранителе она представляет внутреннее состояние объекта в определенный момент. И там, и там «палочка» может иметь сложную внутреннюю структуру, но клиент об этом ничего не «знает». Но даже здесь есть различия. В паттерне команда важную роль играет полиморфизм, поскольку выполнение объекта-команды — полиморфная операция. Напротив, интерфейс хранителя настолько узок, что его можно передавать лишь как значение. Поэтому вполне вероятно, что хранитель не предоставляет полиморфных операций своим клиентам.

ДОЛЖЕН ЛИ ОБМЕН ИНФОРМАЦИЕЙ БЫТЬ ИНКАПСУЛИРОВАННЫМ ИЛИ РАСПРЕДЕЛЕННЫМ?

Паттерны посредник (319) и наблюдатель (339) конкурируют между собой. Различие между ними в том, что наблюдатель распределяет обмен информацией за счет объектов наблюдатель и субъект, а посредник, наоборот, инкапсулирует взаимодействие между другими объектами.

В паттерне наблюдатель участники наблюдатель и субъект должны кооперироваться, чтобы поддержать ограничение. Паттерны обмена информацией определяются тем, как связаны между собой наблюдатели и субъекты; у одного субъекта обычно бывает много наблюдателей, а иногда наблюдатель субъекта сам является субъектом наблюдения со стороны другого объекта. В паттерне посредник ответственность за поддержание ограничения возлагается исключительно на посредника.

На наш взгляд, повторно использовать наблюдателей и субъектов проще, чем посредников. Паттерн наблюдатель способствует разделению и ослаблению связей между наблюдателем и субъектом, что приводит к появлению сравнительно мелких классов, более приспособленных для повторного использования.

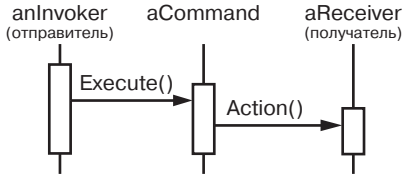
С другой стороны, потоки информации в посреднике проще для понимания, нежели в наблюдателе. Наблюдатели и субъекты обычно связываются вскоре после создания, и понять, каким же образом организована их связь, в последующих частях программы довольно трудно. Если вы знаете паттерн наблюдатель, то понимаете важность того, как именно связаны наблюдатели и субъекты, и представляете, какие связи надо искать. Однако присущая наблюдателю косвенность затрудняет понимание системы.

В языке Smalltalk наблюдатели можно параметризовать сообщениями, применяемыми для доступа к состоянию субъекта, поэтому степень их повторного использования даже выше, чем в C++. Вот почему в Smalltalk паттерн наблюдатель более привлекателен, чем в C++. Следовательно, программист, пишущий на Smalltalk, нередко использует наблюдателя там, где программист на C++ применил бы паттерн посредник.

РАЗДЕЛЕНИЕ ПОЛУЧАТЕЛЕЙ И ОТПРАВИТЕЛЕЙ

Когда взаимодействующие объекты напрямую ссылаются друг на друга, они становятся зависимыми, а это может отрицательно сказаться на повторном использовании системы и разбиении ее на уровни. Паттерны команда, наблюдатель, посредник и цепочка обязанностей указывают разные способы разделения получателей и отправителей запросов. Каждый способ имеет свои достоинства и недостатки.

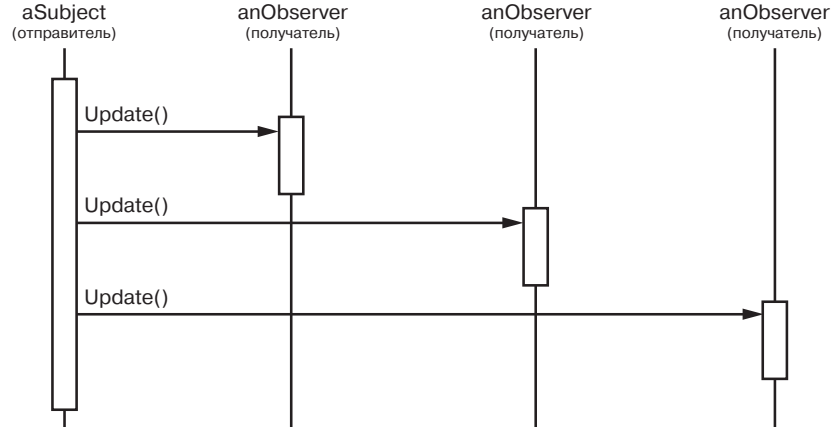
Паттерн команда поддерживает разделение за счет объекта-команды, который определяет привязку отправителя к получателю:



Паттерн команда предоставляет простой интерфейс для выдачи запроса (операцию `Execute`). Определение связи между отправителем и получателем в самостоятельном объекте позволяет отправителю работать с разными получателями. Он отделяет отправителя от получателей, облегчая тем самым повторное использование. Кроме того, объект-команду можно повторно использовать для параметризации получателя различными отправителями. Номинально паттерн команда требует определения подкласса для каждой

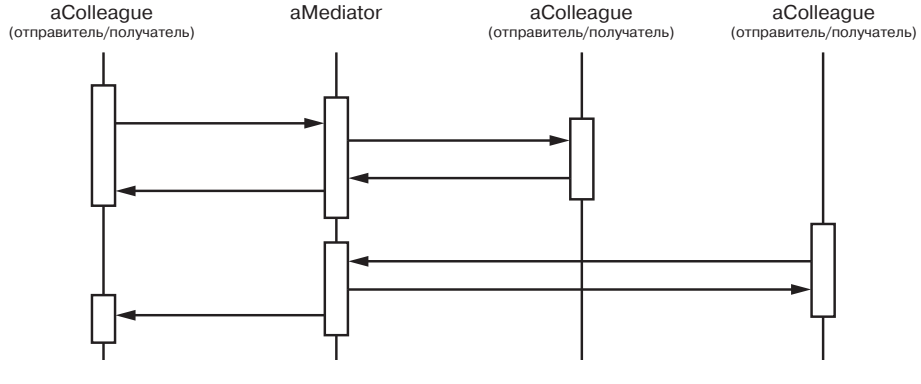
связи «отправитель — получатель», хотя имеются способы реализации, при которых удается избежать порождения подклассов.

Паттерн наблюдатель отделяет отправителей (субъектов) от получателей (наблюдателей) путем определения интерфейса для извещения о происшедших с субъектом изменениях. По сравнению с командой в наблюдателе связь между отправителем и получателем слабее, поскольку у субъекта может быть много наблюдателей и их число даже может меняться во время выполнения.



Интерфейсы субъекта и наблюдателя в паттерне наблюдатель предназначены для передачи информации об изменениях. Стало быть, этот паттерн лучше всего подходит для разделения объектов в случае, когда между ними есть зависимость по данным.

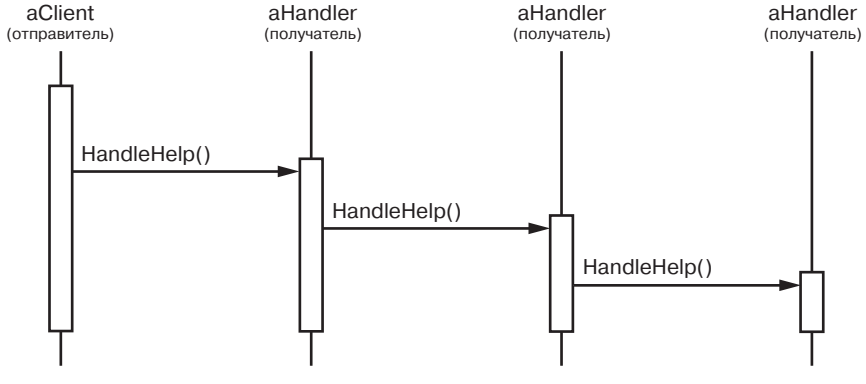
Паттерн посредник разделяет объекты, заставляя их ссылаться друг на друга косвенно, через объект-посредник.



Объект-посредник распределяет запросы между объектами-коллегами и централизует обмен информацией между ними. Таким образом, коллеги могут «общаться» между собой только с помощью интерфейса посредника. Поскольку этот интерфейс фиксирован, посредник может реализовать собственную схему диспетчеризации для большей гибкости. Разрешается кодировать запросы и упаковывать аргументы так, что коллеги смогут запрашивать выполнение операций из заранее неизвестного множества.

Паттерн посредник часто способствует уменьшению числа подклассов в системе, поскольку централизует весь обмен информацией в одном классе, вместо того чтобы распределять его по подклассам. С другой стороны, ситуативные схемы диспетчеризации снижают безопасность типов.

Наконец, паттерн цепочка обязанностей отделяет отправителя от получателя за счет передачи запроса по цепочке потенциальных получателей.



Поскольку интерфейс между отправителями и получателями фиксирован, то цепочка обязанностей также может нуждаться в специализированной схеме диспетчеризации. Поэтому она обладает теми же недостатками с точки зрения безопасности типов, что и посредник. Цепочка обязанностей — хороший способ разделить отправителя и получателя в случае, если она уже является частью структуры системы, а один объект из группы может принять на себя обязанность обработать запрос. Данный паттерн повышает гибкость и за счет того, что цепочку можно легко изменить или расширить.

РЕЗЮМЕ

За немногочисленными исключениями паттерны поведения дополняют и усиливают друг друга. Например, класс в цепочке обязанностей, скорее

всего, будет содержать хотя бы один шаблонный метод (373). Он может пользоваться примитивными операциями, чтобы определить, должен ли объект обработать запрос сам, а также в случае необходимости выбрать объект, которому следует переадресовать запрос. Цепочка может применять паттерн команда для представления запросов в виде объектов. Зачастую интерпретатор (287) пользуется паттерном состояние для определения контекстов синтаксического разбора. Иногда итератор обходит агрегат, а посетитель выполняет операцию с каждым его элементом.

Паттерны поведения хорошо сочетаются и с другими паттернами. Например, система, в которой применяется паттерн компоновщик (196), время от времени использует посетителя для выполнения операций над компонентами, а также задействует цепочку обязанностей, чтобы обеспечить компонентам доступ к глобальным свойствам через их родителя. Бывает, что в системе применяется и паттерн декоратор (209) для переопределения некоторых свойств частей композиции. А паттерн наблюдатель может связать структуры разных объектов, тогда как паттерн состояние позволит компонентам изменять свое поведение при изменении состояния. Сама композиция может быть создана с применением строителя (124) и рассматриваться как прототип (146) какой-то другой частью системы.

Это характерно для хорошо спроектированных объектно-ориентированных систем: внешне они похожи на собрание многочисленных паттернов, но вовсе не потому, что их проектировщики мыслили именно такими категориями. Композиция на уровне паттернов, а не классов или объектов, позволяет добиться той же синергии, но с меньшими усилиями.

ГЛАВА 6

ЗАКЛЮЧЕНИЕ

Возможно, кто-то скажет, что практическая ценность этой книги не так уж велика. Ведь в ней не описываются никакие ранее неизвестные алгоритмы или приемы программирования. Здесь не предлагается строгой методологии проектирования систем, не делается попытки разработать новую теорию проектирования, а всего лишь документируются существующие приемы. А из этого выходит, что данная книга является неплохим руководством начального уровня, но уж опытному специалисту в области объектно-ориентированного проектирования она ни к чему.

Надеемся, вы думаете по-другому. Каталогизация паттернов проектирования важна сама по себе. Она задает стандартные названия и определения тем приемам, которыми мы постоянно пользуемся. Если не изучать паттерны проектирования программ, их нельзя будет и усовершенствовать, а придумывать новые будет сложнее.

Данная книга — только начало. Приведенные в ней паттерны постоянно используются в объектно-ориентированном проектировании, но узнать о них можно только из устной молвы или путем изучения существующих систем. После прочтения ранних вариантов этой книги многие проектировщики стали записывать, какими паттернами они пользуются, а в своей окончательной форме книга побудит к аналогичной работе еще более широкую аудиторию. Хочется верить, что это положит начало движению за документирование опыта практического программирования.

В данной главе обсуждается, какое влияние окажут паттерны на развитие проектирования, как они связаны с другими его сторонами и как самостоятельно включиться в работу по поиску и каталогизации паттернов.

6.1. ЧЕГО ОЖИДАТЬ ОТ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Вот несколько вариантов того, как паттерны, описанные в книге, могут повлиять на ваш подход к проектированию объектно-ориентированных программ. Приведенные соображения основаны на нашем опыте повседневной работы с ними.

ЕДИНЫЙ СЛОВАРЬ ПРОЕКТИРОВАНИЯ

Изучение работы высококвалифицированных программистов, пишущих на традиционных языках, показало, что их знания и опыт основаны не только на синтаксисе языка, но и на более крупных концептуальных структурах, таких как алгоритмы, структуры данных и идиомы [AS85, Cop92, Cur89, SS86], а также планировании шагов по достижению поставленных целей [SE84]. Возможно, они прилагают основные усилия к тому, чтобы найти аналогии текущей задаче в тех планах, алгоритмах, структурах данных и идиомах, которыми они пользовались ранее или о которых было известно.

Специалисты в области информатики выбирают имена и каталогизируют алгоритмы и структуры данных, но зачастую мы не заботимся о том, чтобы как-то назвать другие виды паттернов. Паттерны предоставляют проектировщикам единую терминологию, которой можно пользоваться для общения, документирования и изучения возможных альтернатив. Система начинает выглядеть менее сложной, поскольку появляется возможность говорить о ней на более высоком уровне абстракции, чем нотация языка проектирования или программирования.

После знакомства с паттернами проектирования, описанными в книге, ваш язык проектирования почти наверняка изменится. И вы начнете включать названия паттернов в обсуждения проектов: «Здесь стоит воспользоваться “Наблюдателем”» или «А эти классы можно преобразовать в “стратегию”».

ПОМОЩЬ ПРИ ДОКУМЕНТИРОВАНИИ И ИЗУЧЕНИИ

Знание описанных в книге паттернов проектирования помогает понять существующие объектно-ориентированные системы, в большинстве которых паттерны применяются. Люди, изучающие объектно-ориентированные языки, часто жалуются на то, что наследование в системах используется запутанно, а разобраться в логике передачи управления очень трудно.

Паттерны проектирования также повысят вашу квалификацию как проектировщика. Они предоставляют стандартные решения для типичных задач. Если вы достаточно долго проработаете с объектно-ориентированными системами, то, возможно, освоите описываемые здесь паттерны на собственном опыте. Книга существенно ускорит процесс изучения. Отметим также, что знание паттернов поможет начинающему проектировщику работать так, как работает эксперт.

Понять систему, которая описана в категориях применяемых в ней паттернов проектирования, намного проще. В противном случае для выявления паттернов пришлось бы восстанавливать дизайн по исходным текстам. Наличие единого словаря означает, что вам нет нужды описывать паттерн целиком; достаточно просто назвать его, а читатель поймет, о чем идет речь. Проектировщик, незнакомый с паттернами, должен будет сначала найти информацию о них, но это все равно проще, чем обратное конструирование.

Мы постоянно применяем паттерны в своих проектах и считаем их исключительно полезными. Тем не менее, кто-то сочтет, что наши способы применения паттернов слишком просты: мы используем паттерны при выборе имен для классов, как основу для размышлений и обучения хорошему проектированию, а также для описания проектов [BJ94]. Но легко представить себе и более изощренные способы применения паттернов, например основанные на них CASE-средства или гипертекстовые документы. Впрочем, паттерны окажут значительную помощь и без сложных инструментов.

ДОПОЛНЕНИЕ СУЩЕСТВУЮЩИХ МЕТОДОВ

С помощью объектно-ориентированного проектирования можно создать хороший дизайн, обучить начинающих проектировщиков правильным приемам работы и стандартизовать методики разработки хороших проектов. Обычно метод проектирования определяет обозначения (как правило, графические) для моделирования различных аспектов проекта, а также набор правил, диктующих, как и когда применять каждое обозначение; зачастую удается описать проблемы, с которыми пришлось столкнуться в ходе работы над проектом, способы их разрешения и способы оценки полученного результата. Но опыт квалифицированных разработчиков не исчерпывается одними методами проектирования, которые они используют.

Думается, что наши паттерны оказываются важным дополнением к методам объектно-ориентированного проектирования. Они показывают, как применять такие базовые приемы, как объекты, наследование и полиморфизм, демонстрируют, как можно параметризовать систему алгоритмом, поведением,

состоянием или видом объектов, которые предполагается создавать. Паттерны проектирования позволяют не просто зафиксировать результаты решений, а ответить на многочисленные «почему», возникающие в ходе проектирования. Разделы «Применимость», «Результаты» и «Реализация» в описаниях паттернов помогут вам сориентироваться при принятии решения.

Паттерны проектирования особенно полезны тогда, когда нужно преобразовать аналитическую модель в модель реализации. Вопреки многочисленным заверениям о беспрепятственном переходе от объектно-ориентированного анализа к проектированию, на практике этот процесс никогда не происходит гладко. В гибком проекте, ориентированном на повторное использование, имеются объекты, которых нет в аналитической модели. На проект оказывают влияние выбранные язык программирования и библиотеки классов. Аналитические модели часто приходится пересматривать, чтобы обеспечить повторное использование. Многие паттерны, включенные в каталог, непосредственно связаны с такого рода вопросами, почему мы и называем их паттернами *проектирования*.

Полноценная методика проектирования основывается не только на паттернах проектирования. Тут могут быть паттерны и анализа, и пользовательского интерфейса, и оптимизации производительности. Но паттерны — очень важная составная часть методики проектирования, которая до сих пор отсутствовала.

ЦЕЛЬ РЕФАКТОРИНГА

Повторно используемое программное обеспечение часто приходится подвергать рефакторингу [OJ90]. Паттерны проектирования помогают вам решить, как именно провести рефакторинг, и могут уменьшить вероятность рефакторинга в будущем.

По Брайану Футу (Brian Foote), жизненный цикл объектно-ориентированной программы состоит из трех фаз: *прототипизации*, *экстенсивного роста* и *консолидации* [Foo92].

Фаза прототипизации характеризуется высокой активностью проектировщиков, направленной на то, чтобы программа начала работать. При этом быстро создается прототип, который последовательно изменяется до тех пор, пока не будут решены первоначальные задачи. Обычно на данном этапе программа представляет собой набор иерархий классов, отражающих сущности предметной области. Основной вид повторного использования — это принцип «прозрачного ящика» посредством наследования.

После ввода в эксплуатацию развитие программы определяется двумя противоречивыми факторами: (1) программе необходимо отвечать все новым требованиям и (2) одновременно должна повышаться степень ее повторной используемости. Новые требования диктуют необходимость добавления новых классов и операций, быть может, даже целых иерархий классов. Эти требования могут удовлетворяться, когда начинается фаза экстенсивного роста программы. Тем не менее, данный период не может продолжаться долго. В конце концов, программа становится слишком негибкой и не поддается дальнейшим изменениям. Иерархии классов более не соответствуют одной предметной области. Напротив, они отражают множество разных предметных областей, а классы определяют совершенно разнородные операции и переменные экземпляров.

Чтобы программа могла развиваться и дальше, ее необходимо пересмотреть и реорганизовать; этот процесс и называется *рефакторингом*. Именно в этот период часто формируются каркасы. При рефакторинге классы, описывающие специализированные и универсальные компоненты, отделяются друг от друга, операции перемещаются вверх-вниз по иерархии, а интерфейсы классов становятся более рациональными. В фазе консолидации появляется много новых объектов, часто в результате декомпозиции существующих и использования композиции вместо наследования. Таким образом, «прозрачный ящик» заменяется «черным». В связи с непрекращающимся потоком новых требований и стремлением ко все более высокой степени повторной используемости объектно-ориентированная программа должна вновь и вновь проходить через фазы экстенсивного роста и консолидации.

От этого цикла никуда не деться. Но хороший проектировщик предвидит изменения, которые могут потребовать рефакторинга. Он знает, какие структуры классов и объектов позволят избежать рефакторинга, его проект устойчив к изменениям требований. Тщательный анализ требований поможет выявить требования, которые с большой вероятностью изменятся

на протяжении жизненного цикла программы, и в хорошем проекте все это учитывается.

В наших паттернах проектирования отражены многие структуры, появившиеся в результате рефакторинга. Применение паттернов на ранних стадиях проекта часто предотвращает рефакторинг в будущем. Но даже если вы не увидели возможностей для применения паттерна, пока не создали всю систему, паттерн все равно «подскажет» пути ее возможного изменения.

6.2. КРАТКАЯ ИСТОРИЯ

Начало этому каталогу положила докторская диссертация Эриха [Gam91, Gam92]. Почти половина всех паттернов была представлена в этой работе. К моменту проведения конференции OOPSLA '91 диссертацию официально признали независимым каталогом, и для продолжения работы над ним к Эриху присоединился Ричард. Вскоре после этого к компании примкнул и Джон. Незадолго до конференции OOPSLA '92 в группу влился Ральф. Мы изо всех сил старались, чтобы каталог был готов к публикации в трудах ECOOP '93, но вскоре осознали, что статью на 90 страницах не примут. Поэтому пришлось составить краткий реферат каталога, который и был опубликован. Вскоре после этого мы решили превратить каталог в книгу.

Названия, которые мы присваивали паттернам, по ходу дела менялись. «Обертка» (Wrapper) стала декоратором, «клей» (Glue) — фасадом, «холостяк» (Solitaire) — одиночкой, «бродяга» (Walker) — посетителем. Парочка паттернов осталась за бортом, поскольку мы не сочли их достаточно важными. Но в остальном набор паттернов в каталоге почти не менялся с конца 1992 г. Однако сами паттерны эволюционировали очень сильно.

На самом деле заметить, что нечто представляет собой паттерн, несложно. Мы все четверо активно трудимся над созданием объектно-ориентированных систем и обнаружили, что, когда поработаешь с достаточно большим числом таких систем, выявлять паттерны становится просто. Но *найти* паттерн куда проще, чем *описать* его, тем более так, чтобы проектировщики, незнакомые с ним, поняли назначение этого приема и уяснили, почему он важен.

Если вы строите системы, а затем размышляете над тем, что было сделано, вы увидите паттерны в результатах своей работы. Но паттерны трудно описать так, чтобы незнакомые с ними люди поняли их и осознали их важность. Специалисты уже на самых ранних стадиях осознали ценность каталога. Но понять паттерны смогли лишь те, кто их уже использовал.

Так как одной из главных целей книги было научить объектно-ориентированному проектированию, то мы пришли к выводу о необходимости улучшить каталог. Увеличили средний объем описания одного паттерна с двух до десяти с лишним страниц, включив подробный раздел «Мотивация» и пример кода. Мы также решили рассматривать различные плюсы и минусы, а также разные способы реализации паттернов. В результате изучать паттерны стало легче.

Еще одно важное изменение, внесенное не так давно: задаче, которую призван решить тот или иной паттерн, стало уделяться более пристальное внимание. Проще взглянуть на паттерн как на решение, а не как на прием, который можно приспособить под собственные нужды и повторно использовать. Труднее увидеть, когда паттерн *подходит* — то есть охарактеризовать те задачи, которые он решает, и тот контекст, в котором он является наилучшим вариантом. Проще разглядеть, *что* делается, чем понять, *почему* так делается. Понимать назначение паттерна тоже важно, поскольку это помогает определить, какой паттерн стоит применить. Также паттерны помогают понять структуру существующих систем. Автор паттерна должен определить и охарактеризовать проблему, которую решает паттерн, даже если он сделает это «задним числом» после обнаружения решения.

6.3. ПРОЕКТИРОВЩИКИ ПАТТЕРНОВ

Мы не единственные, кому интересно писать книги, которые содержат каталог паттернов, применяемых специалистами. Мы — часть обширного сообщества, заинтересованного в паттернах вообще и в паттернах, имеющих отношение к программному обеспечению, в частности. Кристофер Александр — это архитектор, который первым начал изучать паттерны в строительстве и разработал «язык паттернов» для их генерирования. Работа Александра постоянно вдохновляла нас. Поэтому уместно и поучительно сравнить его и наши старания. Затем мы обратимся к работам других авторов по паттернам, связанным с программным обеспечением.

ЯЗЫКИ ПАТТЕРНОВ АЛЕКСАНДРА

Наша работа напоминает работу Александра во многих отношениях. Обе основаны на изучении существующих систем и нахождении в них паттернов. И там, и там есть шаблоны для описания паттернов, хотя и совершенно различные. Работы основаны на естественном языке и примерах,

а не на формальных языках. В обеих для каждого паттерна приводятся обоснования.

Но во многих отношениях наши работы различаются:

- люди строят здания много тысяч лет, так что существует множество классических примеров. Программные же системы мы начали создавать сравнительно недавно, и лишь немногие признаны классикой;
- Александр приводит порядок, в котором следует использовать его паттерны, мы — нет;
- в паттернах Александра упор сделан на проблемах, в то время как паттерны проектирования гораздо подробнее описывают решение;
- Александр утверждает, что его паттерны способны сгенерировать проект всего здания. Мы не считаем, что наши паттерны могут создать законченную программу.

Когда Александр говорит, что можно спроектировать здание, просто применяя паттерны один за другим, то он преследует те же цели, что и методисты объектно-ориентированного проектирования, которые приводят пошаговые правила. Александр не отрицает творческого подхода; некоторые из его паттернов требуют понимания привычек и обычаев людей, которые будут жить в здании, а его вера в «поэзию» проектирования подразумевает, что уровень проектировщика отнюдь не должен ограничиваться владением языком[1]. Но из его описания того, как паттерны генерируют проект, следует, что язык способен сделать процесс проектирования детерминированным и повторяемым.

Точка зрения Александра помогла нам сместить акцент на компромиссы проектирования — различные «силы», которые формируют дизайн. Под влиянием этого человека мы упорно трудились над тем, чтобы понять применимость и последствия каждого из наших паттернов. Идеология работы Александра уберегла нас от волнений по поводу формального представления паттернов. Хотя такое представление, возможно, помогло бы автоматизировать паттерны, мы полагаем, что на данном этапе важнее исследовать пространство паттернов проектирования, а не формализовать его.

С точки зрения Александра паттерны, описанные в книге, не составляют языка. Принимая во внимание многообразие программных систем, трудно представить себе, как предложить «полный» набор паттернов, из которого можно было бы вывести пошаговые инструкции по созданию приложения.

---

[1] См. «The poetry of the language» [AIS+77].

Для некоторых классов приложений (скажем, систем генерирования отчетов или ввода данных путем заполнения форм) это возможно. Но наш каталог представляет собой лишь набор взаимосвязанных паттернов, мы не пытаемся выдать его за *язык* паттернов.

Вряд ли когда-либо будет создан полный язык паттернов для проектирования программ. Но, безусловно, можно создать каталог *более полный*, чем наш. В него можно было бы включить описание каркасов и способов их применения [Joh92], паттернов проектирования пользовательского интерфейса [BJ94], паттернов анализа [Coa92] и прочих аспектов разработки программ. Паттерны проектирования — это всего лишь часть более широкого языка паттернов в программном обеспечении.

ПАТТЕРНЫ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Нашим первым коллективным опытом изучения архитектуры программного обеспечения было участие в семинаре OOPSLA '91, который проводил Брюс Андерсон (Bruce Anderson). Семинар был посвящен составлению справочника для архитекторов программных систем. Данное событие положило начало целой серии встреч, последняя из которых состоялась в августе 1994 г. на первой конференции по языкам паттернов программ. В результате сформировалось сообщество людей, заинтересованных в документировании опыта разработки программного обеспечения.

Разумеется, ту же цель видели перед собой и другие исследователи. Книга Дональда Кнута «Искусство программирования для ЭВМ» [Knu73] была одной из первых попыток систематизировать знания, накопленные при разработке программ, хотя акцент в ней был сделан на описании алгоритмов. Но даже эта задача оказалась слишком трудной, так что работа осталась незаконченной. Серия книг Graphics Gems [Gla90, Arv91, Kir92] — еще один каталог, посвященный проектированию, хотя и он в основном посвящен алгоритмам. Программа Domain Specific Software Architecture (Архитектура проблемно-ориентированного программного обеспечения), которую спонсирует Министерство обороны США [GM92], направлена на подбор информации архитектурного плана. Исследователи, занятые разработкой баз знаний, стремятся отразить накопленный опыт разработок. Есть и много других групп, задачи которых в той или иной мере сходны с нашими.

Большое влияние на нас также оказала книга Джеймса Коплиена «Advanced C++: Programming Styles and Idioms» [Cop92]. Описанные в ней паттерны в большей степени, чем наши, ориентированы на C++. Кроме того, в книге

приводится много низкоуровневых паттернов. Коплиен всегда был активным членом сообщества проектировщиков, заинтересованных в паттернах. Сейчас он работает над паттернами, описывающими роли людей в организациях, занятых разработкой программ.

Одним из первых, кто стал популяризировать работы Кристофера Александра среди программистов, был Кент Бек (Kent Beck). В 1993 г. он начал вести колонку в журнале The Smalltalk Report, посвященную паттернам в языке Smalltalk. Некоторое время паттерны собирал Питер Коад (Peter Coad). В основном в его работе [Coa92] представлены паттерны анализа. Мы не видели последние паттерны, разработанные им, хотя и слышали, что он над ними работает. Также ходят слухи о нескольких книгах на тему паттернов, над которыми сейчас работают авторы, но не видели ни одной из них. Все, что мы можем — сказать, что их появление ожидается в будущем.

6.4. ПРИГЛАШЕНИЕ

Что можете сделать лично вы, если вас интересуют паттерны? Прежде всего применяйте их и ищите другие паттерны, которые лучше отражают ваш подход к проектированию. Разработайте свой словарь паттернов и используйте его, в частности, в беседах с коллегами о своих проектах. Размышляя о проектах и описывая их, не забывайте о словаре.

Во-вторых, относитесь к материалу критично! Каталог паттернов — это плод напряженной работы, не только нашей, но и десятков рецензентов, который делились своими замечаниями. Если вы наткнулись на какую-то проблему или полагаете, что объяснения должны быть более подробными, пишите нам. То же самое относится к любому каталогу паттернов: авторам нужна обратная связь с читателями! Одна из самых полезных особенностей паттернов состоит в том, что они выводят процесс принятия проектных решений из туманной области интуиции. С помощью паттернов авторы могут явно сформулировать, на какие компромиссы им пришлось идти. А это, в свою очередь, помогает разглядеть, каковы же минусы их паттернов, и вступить в осмысленную дискуссию. Не упускайте такой возможности.

В-третьих, выявляйте паттерны, которыми пользуетесь, и фиксируйте их. Включите их в состав документации по своей программе. Вовсе не обязательно работать в научно-исследовательской лаборатории, чтобы находить паттерны. Не стесняйтесь составить свой собственный каталог паттернов, но... пусть кто-нибудь поможет вам облечь его в достойную форму!

6.5. НА ПРОЩАНИЕ

В лучших проектах используется много паттернов проектирования. Единое целое образуется в результате их согласованных взаимных действий. Вот что говорит об этом Кристофер Александр: «Можно строить здания, нанизывая паттерны в достаточно произвольном порядке. Такое здание будет просто собранием паттернов. В нем нет плотности. Нет основательности. Но можно объединять паттерны и так, что в одном и том же физическом объеме они будут перекрывать друг друга. Тогда здание получается очень плотным, в небольшом пространстве сосредоточивается много функций. За счет такой плотности здание приобретает основательность» (A Pattern Language [AIS+77, стр. xli]).

ПРИЛОЖЕНИЕ А

ГЛОССАРИЙ

Абстрактная операция — операция, которая объявляет сигнатуру, но не реализует ее. В C++ абстрактные операции соответствуют *чисто виртуальным функциям* классов.

Абстрактная связанность — говорят, что класс A абстрактно связан с абстрактным классом B, если в A есть ссылка на B. Такое отношение мы называем абстрактной связанностью, поскольку A ссылается на тип объекта, а не на конкретный объект.

Абстрактный класс — класс, единственным назначением которого является определение интерфейса. Абстрактный класс полностью или частично делегирует свою реализацию подклассам. Создавать экземпляры абстрактного класса нельзя.

Агрегированный объект — объект, составленный из подобъектов. Подобъекты называются частями агрегата, и агрегат отвечает за них.

Делегирование — механизм реализации, при котором объект перенаправляет или делегирует запрос другому объекту (уполномоченному). Уполномоченный выполняет запрос от имени исходного объекта.

Деструктор — в C++ это операция, которая автоматически вызывается для очистки объекта непосредственно перед его удалением.

Динамическое связывание — ассоциация между запросом к объекту и одной из его операций, устанавливаемая во время выполнения. В C++ динамически связываться могут только виртуальные функции.

Дружественный класс — в C++: класс, обладающий теми же правами доступа к операциям и данным некоторого класса, что и сам этот класс.

Закрытое наследование — в C++: класс, наследуемый только ради реализации.

Замещение — переопределение операции, унаследованной от родительского класса, в подклассе.

Инкапсуляция — результат сокрытия представления и реализации в объекте. Представление невидимо и недоступно извне. Получить доступ к представлению объекта и модифицировать его можно только с помощью операций.

Инструментальная библиотека (toolkit) — набор классов, обеспечивающих полезную функциональность, но не определяющих дизайн приложения.

Интерфейс — набор всех сигнатур, определенных операциями объекта. Интерфейс описывает множество запросов, на которые может отвечать объект.

Каркас — набор взаимодействующих классов, описывающих повторно применимый дизайн некоторой категории программ. Каркас задает архитектуру приложения, разбивая его на отдельные классы с четко определенными функциями и взаимодействиями. Разработчик настраивает каркас под конкретное приложение путем порождения подклассов и составления композиций из объектов, принадлежащих классам каркаса.

Класс — определяет интерфейс и реализацию объекта. Описывает внутреннее представление и операции, которые объект может выполнять.

Композиция объектов — объединение нескольких объектов для получения более сложного поведения.

Конкретный класс — класс, в котором нет абстрактных операций. Может иметь экземпляры.

Конструктор — в C++: операция, автоматически вызывающаяся для инициализации новых экземпляров.

Метакласс — в Smalltalk классы являются объектами. Метакласс — это класс объекта-класса.

Наследование — отношение, которое определяет одну сущность в терминах другой. В случае наследования класса новый класс определяется в терминах одного или нескольких родительских классов. Новый класс наследует интерфейс и реализацию от своих родителей. Новый класс называется подклассом или производным классом (в C++). Наследование класса объединяет наследование интерфейса и наследование реализации.

В случае наследования интерфейса новый интерфейс определяется в терминах одного или нескольких существующих. При наследовании реализации новая реализация определяется в терминах одной или нескольких существующих.

Объект — имеющаяся во время выполнения сущность, в которой хранятся данные и процедуры для работы с ними.

Операция — на данные объекта можно воздействовать только с помощью его операций. Объект выполняет операцию, когда получает запрос. В C++ операции называются *функциями класса*, в Smalltalk — *методами*.

Операция класса — операция, определенная для класса в целом, а не для индивидуального объекта. В C++ операции класса называются *статическими функциями*.

Отношение агрегирования — отношение агрегата и его частей. Класс определяет такое отношение для своих экземпляров, то есть агрегированных объектов.

Отношение осведомленности — говорят, что одному классу известно о другом, если первый ссылается на второй.

Параметризованный тип — тип, где некоторые составляющие типы оставлены неопределенными. Они передаются как параметры в точке использования. В C++ параметризованные типы называются шаблонами.

Паттерн проектирования — паттерн проектирования именует, мотивирует и объясняет конкретный прием проектирования, который относится к задаче, часто возникающей при работе над объектно-ориентированными системами. Паттерн описывает задачу, ее решение, область применимости этого решения и его результаты. Он также содержит рекомендации по реализации и примеры. Под решением понимается схема организации объектов и классов, позволяющая справиться с проблемой. Паттерн адаптируется для работы в конкретных условиях и реализуется в заданном контексте.

Переменная экземпляра — элемент данных, определяющий часть представления объекта. В C++ используется термин *переменная класса*.

Подкласс — класс, наследующий другому классу. В C++ подкласс называется *производным классом*.

Подсистема — независимая группа классов, функционирующих совместно для выполнения набора обязанностей.

Подтип — один тип называется подтипом другого, если интерфейс первого содержит интерфейс второго.

Полиморфизм — способность подставлять во время выполнения вместо одного объекта другой с совместимым интерфейсом.

Получатель — объект, которому направлен запрос.

Примесь — класс, спроектированный так, чтобы сочетаться с другими классами путем наследования. Примеси-классы обычно абстрактны.

Прозрачный ящик как способ повторного использования — стиль повторного использования, основанный на наследовании классов. Подкласс повторно использует интерфейс и реализацию родительского класса, но может также иметь доступ к закрытым для других аспектам своего родителя.

Протокол — расширяет концепцию интерфейса за счет включения допустимой последовательности запросов.

Родительский класс — класс, которому наследует другой класс. Синонимы: *суперкласс* (Smalltalk), *базовый класс* (C++) и *класс-предок*.

Связанность — степень зависимости компонентов программы друг от друга.

Сигнатура — под сигнатурой операции понимается сочетание ее имени, параметров и возвращаемого значения.

Ссылка на объект — значение, которое идентифицирует другой объект.

Супертип — тип родителя, которому наследует данный тип.

Схема взаимодействий — схема, на которой показан поток запросов между объектами.

Схема классов — схема, на которой изображены классы, их внутренняя структура и операции, а также статические связи между ними.

Схема объекта — схема, на которой изображена структура конкретного объекта во время выполнения.

Тип — имя конкретного интерфейса.

Черный ящик как способ повторного использования — стиль повторного использования, основанный на композиции объектов. Объекты-компоненты не раскрывают друг другу деталей своего внутреннего устройства и потому могут быть уподоблены черным ящикам.

ПРИЛОЖЕНИЕ Б

ОБЪЯСНЕНИЕ НОТАЦИИ

На протяжении всей книги мы пользуемся схемами для иллюстрации важных идей. Некоторые схемы нестандартны (например, снимок экрана, где изображено диалоговое окно, или схематичное изображение дерева объектов). Но при описании паттернов проектирования для обозначения отношений и взаимодействий между классами и объектами применяется более формальная нотация. В настоящем приложении эта нотация рассматривается подробно.

Мы пользуемся тремя видами схем:

- на *схеме классов* отображены классы, их структура и статические отношения между ними;
- на *схеме объектов* показана структура объектов во время выполнения;
- на *схеме взаимодействий* изображен поток запросов между объектами.

В описании каждого паттерна проектирования есть хотя бы одна схема классов. Остальные используются, если в них возникает необходимость. Схема классов и объектов основаны на методологии OMT (Object Modeling Technique — методика моделирования объектов) [RBP+91, Rum94][1]. Схема взаимодействий заимствованы из методологии Objectory [JCJO92] и метода Буча [Boo94].

[1] В OMT для обозначения схем классов используется термин «схема объектов». Мы же зарезервировали термин «схема объекта» исключительно для описания структуры объекта.

Б.1. СХЕМА КЛАССОВ

На рис. В.1а представлена нотация OMT для абстрактных и конкретных классов. Класс обозначается прямоугольником, в верхней части которого жирным шрифтом напечатано имя класса. Основные операции класса перечисляются под именем класса. Все переменные экземпляра располагаются ниже операций. Информация о типе необязательна; мы пользуемся синтаксисом C++, ставя имя типа перед именем операции (для обозначения типа возвращаемого значения), переменной экземпляра или фактического параметра. Курсив служит указанием на то, что класс или операция абстрактны.

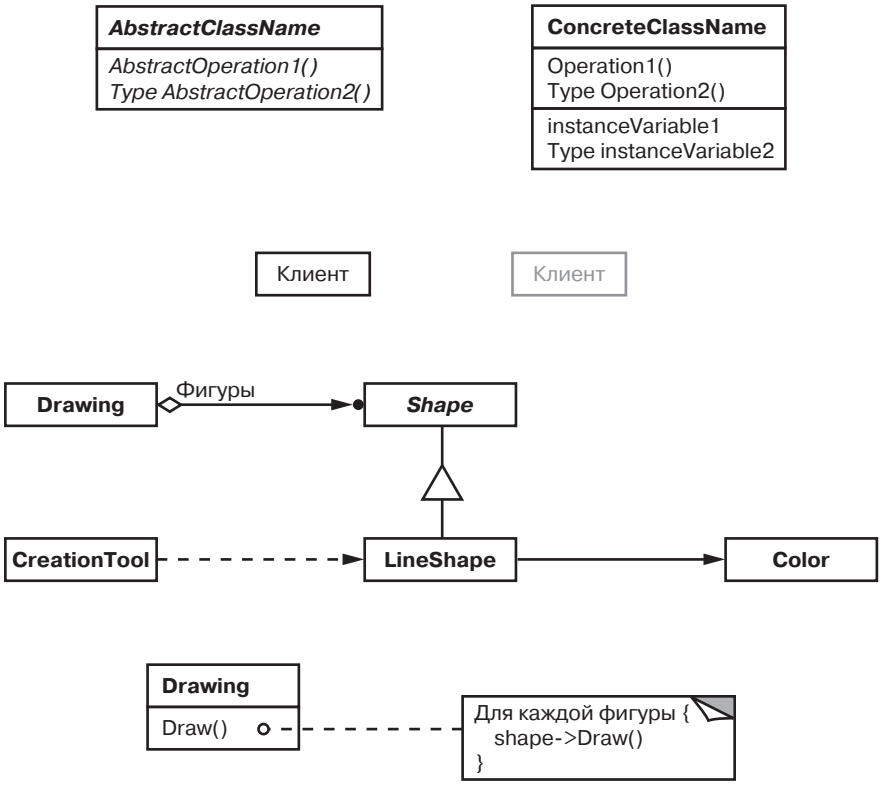


Рис. Б.1. Нотация схем классов: а) абстрактные и конкретные классы; б) класс клиента участника (слева) и класс неявного клиента (справа); в) отношения между классами; г) аннотация на псевдокоде

При использовании некоторых паттернов проектирования полезно видеть, где классы клиентов ссылаются на классы-участники. Если паттерн включает класс клиента в качестве одного из участников (это означает, что на клиента возлагаются определенные функции), то клиент изображается как обычный класс. Так, например, обстоит дело в паттерне приспособленец (231). Если же клиент не входит в состав участников паттерна (то есть не несет никаких обязанностей), то его изображение все равно полезно, поскольку проясняет способ взаимодействия участников с клиентами. В этом случае классы клиентов изображаются бледным шрифтом, как показано на рис. Б.1б. Примером может служить паттерн заместитель (246). Бледный шрифт клиента напоминает также о том, что мы специально не включили клиента в состав участников.

На рис. Б.1в показаны отношения между классами. В нотации OMT для обозначения наследования классов используется треугольник, направленный от подкласса (на рисунке — `LineShape`) к родительскому классу (`Shape`). Ссылка на объект, представляющая отношение агрегирования «является частью», обозначается линией со стрелкой с ромбиком на конце. Стрелка указывает на агрегируемый класс (например, `Shape`). Линия со стрелкой без ромбика обозначает отношение осведомленности (так, `LineShape` содержит ссылку на объект `Color`, который может использоваться также и другими фигурами). Рядом с началом стрелки может находиться еще и имя ссылки, позволяющее отличить ее от других ссылок[1].

Еще одно полезное свойство, которое следует визуализировать, — то, какие классы создают экземпляры других классов. Для этого используется пунктирная линия, поскольку OMT такого отношения не поддерживает. Мы называем их *отношениями «создает»*. Стрелка направлена в сторону класса, экземпляр которого создается. На рис. Б.1в класс `CreationTool` создает объекты класса `LineShape`.

---

[1] В OMT определены также ассоциации между классами, изображаемые простыми линиями, соединяющими прямоугольники классов. Хотя на стадии анализа ассоциации полезны, нам кажется, что их уровень слишком высок для выражения отношений в паттернах проектирования, просто потому, что на стадии проектирования ассоциациям следует сопоставить ссылки на объекты или указатели. Ссылки на объекты по сути своей являются направленными и потому лучше подходят для визуализации интересующих нас отношений. Например, классу Drawing (рисунок) известно о классах Shape (фигура), но сами фигуры ничего не «знают» о рисунке, в который они погружены. Выразить такое отношение только лишь с помощью ассоциаций невозможно.

В OMT также определено условное обозначение заполненного круга, обозначающее «более одного». Если такой кружок появляется рядом со стрелкой, то он говорит о том, что она ссылается на несколько объектов или что несколько объектов агрегируются. Рис. Б.1в показывает, что класс `Drawing` агрегирует несколько объектов типа `Shape`.

Наконец, мы дополнили OMT аннотациями на псевдокоде, которые позволяют коротко описать реализацию операций. На рис. Б.1г приведена такая аннотация для операции `Draw` в классе `Drawing`.

Б.2. СХЕМА ОБЪЕКТОВ

На схеме объектов представлены только экземпляры. На ней показан мгновенный снимок объектов в паттерне проектирования. Объектам присваиваются имена вида «`aSomething`», где `Something` — это класс объекта. Для обозначения объекта используется прямоугольник с закругленными углами (что несколько отличается от стандарта OMT), в котором имя объекта отделено от ссылок на другие объекты горизонтальной линией. Стрелки ведут к объектам, на которые ссылается данный объект. На рис. Б.2 изображен соответствующий пример.

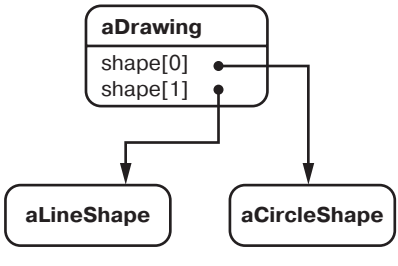


Рис. Б.2. Нотация схем объектов

Б.3. СХЕМА ВЗАИМОДЕЙСТВИЙ

Порядок исполнения запросов, которые объекты посылают друг другу, показан на схеме взаимодействий. Так, на рис. Б.3 представлено, как фигура добавляется к рисунку.

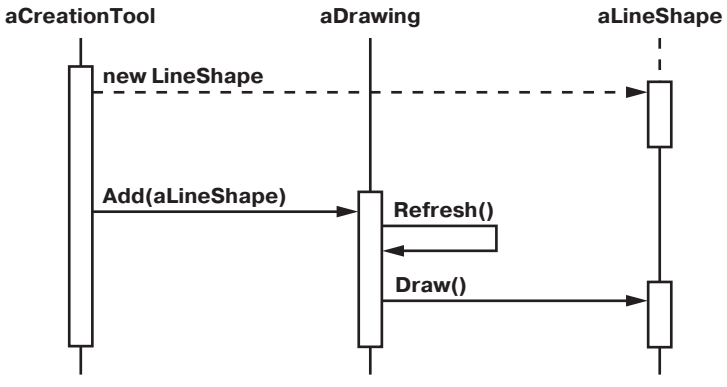


Рис. Б.3. Нотация схем взаимодействий

На схеме взаимодействий время отсчитывается сверху вниз. Сплошная вертикальная линия обозначает время жизни объекта. Соглашение о выборе имен объектов такое же, как на схемах объектов: имени класса предшествует префикс «a» (например, `aShape`). Если объект еще не создан к начальному моменту времени, представленному на схеме, то его вертикальная линия идет пунктиром вплоть до момента создания.

Вертикальный прямоугольник говорит о том, что объект активен, то есть обрабатывает некоторый запрос. Операция может посылать запросы другим объектам, они изображаются горизонтальной линией, указывающей на объект-получатель.

Имя запроса показывается над стрелкой. Запрос на создание объекта представлен пунктирной линией со стрелкой. Запрос объекта-отправителя самому себе изображается стрелкой, указывающей на сам этот объект.

На рис. Б.3 видно, что первый запрос, исходящий от `aCreationTool`, преследует целью создание объекта `aLineShape`. Затем `aLineShape` добавляется к объекту `aDrawing` с помощью операции `Add`, после чего `aDrawing` посылает самому себе запрос на обновление `Refresh`. Отметим, что частью операции `Refresh` является отправка объектом `aDrawing` запроса к `aLineShape`.

ПРИЛОЖЕНИЕ В

ФУНДАМЕНТАЛЬНЫЕ КЛАССЫ

В данном приложении документированы фундаментальные классы, которые применялись нами в примерах кода на C++ в описаниях различных паттернов проектирования. Мы специально стремились сделать эти классы простыми и минимальными. Будут описаны следующие классы:

- `List` — упорядоченный список объектов;
- `Iterator` — интерфейс для последовательного доступа к объектам в агрегате;
- `ListIterator` — итератор для обхода списка;
- `Point` — точка с двумя координатами;
- `Rect` — прямоугольник, стороны которого параллельны осям координат.

Некоторые появившиеся сравнительно недавно стандартные типы C++, возможно, реализованы еще не во всех компиляторах. В частности, если ваш компилятор не поддерживает тип `bool`, его можно определить самостоятельно:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

B.1. LIST

Шаблон класса `List` представляет собой базовый контейнер для хранения упорядоченного списка объектов. В списке хранятся значения элементов, то

есть он пригоден как для встроенных типов, так и для экземпляров классов. Например, запись `List<int>` объявляет список целых int. Но в большинстве паттернов в списке хранятся указатели на объекты, скажем, `List<Glyph*>`. Это позволяет использовать класс `List` для хранения разнородных объектов (точнее, указателей на них).

Для удобства в классе `List` есть синонимы для операций со стеком. Это позволяет явно использовать список в роли стека, не определяя дополнительного класса:

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);
    List(List&);
    ~List();
    List& operator=(const List&);

    long Count() const;
    Item& Get(long index) const;
    Item& First() const;
    Item& Last() const;
    bool Includes(const Item&) const;
    void Append(const Item&);
    void Prepend(const Item&);

    void Remove(const Item&);
    void RemoveLast();
    void RemoveFirst();
    void RemoveAll();

    Item& Top() const;
    void Push(const Item&);
    Item& Pop();
};
```

В следующих разделах операции описываются более подробно.

Конструктор, деструктор, инициализация и присваивание

`List(long size)` — инициализирует список. Параметр `size` определяет начальное число элементов в списке.

`List(List&)` — замещает определяемый по умолчанию копирующий конструктор для правильной инициализации данных.

`~List()` — освобождает внутренние структуры данных списка, но не элементы списка. Не предполагается, что у этого класса будут производные, поэтому деструктор не объявлен виртуальным.

`List& operator=(const List&)` — реализует операцию присваивания.

Обращения к элементам

Следующие операции предназначены для обращения к элементам списка.

`long Count() const` — возвращает число объектов в списке.

`Item& Get(long index) const` — возвращает объект с заданным индексом.

`Item& First() const` — возвращает первый объект в списке.

`Item& Last() const` — возвращает последний объект в списке.

Добавление

`void Append(const Item&)` — добавляет свой аргумент в конец списка.

`void Prepend(const Item&)` — добавляет свой аргумент в начало списка.

Удаление

`void Remove(const Item&)` — удаляет заданный элемент из списка. Для применения этой операции требуется, чтобы тип элементов поддерживал оператор проверки равенства `==`.

`void RemoveFirst()` — удаляет первый элемент из списка.

`void RemoveLast()` — удаляет последний элемент из списка.

`void RemoveAll()` — удаляет все элементы из списка.

Интерфейс стека

`Item& Top() const` — возвращает элемент, находящийся на вершине стека.

`void Push(const Item&)` — заносит элемент в стек.

`Item& Pop()` — извлекает элемент с вершины стека.

B.2. ITERATOR

`Iterator` — это абстрактный класс, который определяет интерфейс обхода агрегата:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Операции класса:

`virtual void First()` — позиционирует итератор на первый объект в агрегате.

`virtual void Next()` — позиционирует итератор на следующий по порядку объект.

`virtual bool IsDone() const` — возвращает `true`, если больше не осталось объектов.

`virtual Item CurrentItem() const` — возвращает объект, находящийся в текущей позиции.

B.3. LISTITERATOR

`ListIterator` реализует интерфейс класса `Iterator` для обхода списка `List`. Его конструктор получает в аргументе список, который нужно обойти:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
```

```
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
};
```

B.4. POINT

Класс `Point` представляет точку на плоскости в декартовых координатах и поддерживает минимальный набор арифметических операций над векторами. Координаты точки определяются так:

```
typedef float Coord;
```

Операции класса `Point` не нуждаются в пояснениях:

```
class Point {
public:
    static const Point Zero;

    Point(Coord x = 0.0, Coord y = 0.0);

    Coord X() const; void X(Coord x);
    Coord Y() const; void Y(Coord y);

    friend Point operator+(const Point&, const Point&);
    friend Point operator-(const Point&, const Point&);
    friend Point operator*(const Point&, const Point&);
    friend Point operator/(const Point&, const Point&);

    Point& operator+=(const Point&);
    Point& operator-=(const Point&);
    Point& operator*=(const Point&);
    Point& operator/=(const Point&);

    Point operator-();

    friend bool operator==(const Point&, const Point&);
    friend bool operator!=(const Point&, const Point&);

    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
};
```

Статическая переменная `Zero` представляет начало координат `Point(0, 0)`.

B.5. RECT

Класс `Rect` представляет прямоугольник, стороны которого параллельны осям координат. Прямоугольник определяется начальной вершиной и размерами, то есть шириной и высотой. Операции класса `Rect` не нуждаются в пояснениях:

```
class Rect {
public:
    static const Rect Zero;

    Rect(Coord x, Coord y, Coord w, Coord h);
    Rect(const Point& origin, const Point& extent);

    Coord Width() const;   void Width(Coord);
    Coord Height() const;  void Height(Coord);
    Coord Left() const;    void Left(Coord);
    Coord Bottom() const;  void Bottom(Coord);

    Point& Origin() const; void Origin(const Point&);
    Point& Extent() const; void Extent(const Point&);

    void MoveTo(const Point&);
    void MoveBy(const Point&);

    bool IsEmpty() const;
    bool Contains(const Point&) const;
};
```

Статическая переменная `Zero` представляет вырожденный прямоугольник:

```
Rect(Point(0, 0), Point(0, 0));
```

БИБЛИОГРАФИЯ

[Add94] Addison-Wesley, Reading, MA. NEXTSTEP General Reference: Release 3, Volumes 1 and 2, 1994.

[AG90] D.B. Anderson and S. Gossain. Hierarchy evolution and the software lifecycle. In TOOLS '90 Conference Proceedings, pages 41–50, Paris, June 1990. Prentice Hall.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977.

[App89] Apple Computer, Inc., Cupertino, CA. Macintosh Programmers Workshop Pascal 3.0 Reference, 1989.

[App92] Apple Computer, Inc., Cupertino, CA. Dylan. An object-oriented dynamic language, 1992.

[Arv91] James Arvo. Graphics Gems II. Academic Press, Boston, MA, 1991.

[AS85] B. Adelson and E. Soloway. The role of domain experience in software design. IEEE Transactions on Software Engineering, 11(11):1351–1360, 1985.

[BE93] Andreas Birrer and Thomas Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In European Conference on Object-Oriented Programming, pages 21–35, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In European Conference on Object-Oriented Programming, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag.

[Boo94] Grady Booch. Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.

[Bor81] A. Borning. The programming language aspects of ThingLab—a constraint-oriented simulation laboratory. ACM Transactions on Programming Languages and Systems, 3(4):343–387, October 1981.

[Bor94] Borland International, Inc., Scotts Valley, CA. A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5, 1994.

[BV90] Grady Booch and Michael Vilot. The design of the C++ Booch components. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 1–11, Ottawa, Canada, October 1990. ACM Press.

[Cal93] Paul R. Calder. Building User Interfaces with Lightweight Objects. PhD thesis, Stanford University, 1993.

[Car89] J. Carolan. Constructing bullet-proof classes. In Proceedings C++ at Work "89. SIGS Publications, 1989.

[Car92] Tom Cargill. C++ Programming Style. Addison-Wesley, Reading, MA, 1992.

[CIRM93] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madeany. Designing and implementing Choices: An object-oriented system in C++. Communications of the ACM, 36(9):117–126, September 1993.

[CL90] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In ACM User Interface Software Technologies Conference, pages 92–101, Snowbird, UT, October 1990.

[CL92] Paul R. Calder and Mark A. Linton. The object-oriented implementation of a document editor. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 154–165, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[Coa92] Peter Coad. Object-oriented patterns. Communications of the ACM, 35(9):152–159, September 1992.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 1–15, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[Cop92] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, 1992.

[Cur89] Bill Curtis. Cognitive issues in reusing software artifacts. In Ted J. Biggerstaff and Alan J. Perlis, editors, Software Reusability, Volume II: Applications and Experience, pages 269–287. Addison-Wesley, Reading, MA, 1989.

[dCLF93] Dennis de Champeaux, Doug Lea, and Penelope Faure. Object-Oriented System Development. Addison-Wesley, Reading, MA, 1993.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors, Software Reusability, Volume II: Applications and Experience, pages 57–71. Addison-Wesley, Reading, MA, 1989.

[Ede92] D. R. Edelson. Smart pointers: They're smart, but they're not pointers. In Proceedings of the 1992 USENIX C++ Conference, pages 1–19, Portland, OR, August 1992. USENIX Association.

[EG92] Thomas Eggenschwiler and Erich Gamma. The ET++SwapsManager: Using object technology in the financial engineering domain. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 166–178, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA, 1990.

[Foo92] Brian Foote. A fractal model of the lifecycles of reusable objects. OOPSLA '92 Workshop on Reuse, October 1992. Vancouver, British Columbia, Canada.

[GA89] S. Gossain and D.B. Anderson. Designing a class hierarchy for domain representation and reusability. In TOOLS '89 Conference Proceedings, pages 201–210, CNIT Paris—La Defense, France, November 1989. Prentice Hall.

[Gam91] Erich Gamma. Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools (in German). PhD thesis, University of Zurich Institut für Informatik, 1991.

[Gam92] Erich Gamma. Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools (in German). Springer-Verlag, Berlin, 1992.

[Gla90] Andrew Glassner. Graphics Gems. Academic Press, Boston, MA, 1990.

[GM92] M. Graham and E. Mettala. The Domain-Specific Software Architecture Program. In Proceedings of DARPA Software Technology Conference, 1992, pages 204–210, April 1992. Also published in CrossTalk, The Journal of Defense Software Engineering, pages 19–21, 32, October 1992.

[GR83] Adele J. Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA, 1983.

[HHMV92] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics, pages 1–22, Champéry, Switzerland, October 1992. Also available as IBM Research Division Technical Report RC 18524 (79392).

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Object-oriented development. IEEE Software, 4(5):71–79, September 1987.

[ION94] IONA Technologies, Ltd., Dublin, Ireland. Programmer's Guide for Orbix, Version 1.2, 1994.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. Object-Oriented Software Engineering—A Use Case Driven Approach. Addison-Wesley, Wokingham, England, 1992.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. Journal of Object-Oriented Programming, 1(2):22–35, June/July 1988.

[JML92] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation, pages 255–274, Dagstuhl, Germany, 1992. Springer-Verlag.

[Joh92] Ralph Johnson. Documenting frameworks using patterns. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 63–76, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[JZ91] Ralph E. Johnson and Jonathan Zweig. Delegation in C++. Journal of Object-Oriented Programming, 4(11):22–35, November 1991.

[Kir92] David Kirk. Graphics Gems III. Harcourt, Brace, Jovanovich, Boston, MA, 1992.

[Knu73] Donald E. Knuth. The Art of Computer Programming, Volumes 1, 2, and 3. Addison-Wesley, Reading, MA, 1973.

[Knu84] Donald E. Knuth. The TEXbook. Addison-Wesley, Reading, MA, 1984.

[Kof93] Thomas Kofler. Robust iterators in ET++. Structured Programming, 14:62–85, March 1993.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3):26–49, August/September 1988.

[LaL94] Wilf LaLonde. Discovering Smalltalk. Benjamin/Cummings, Redwood City, CA, 1994.

[LCI+92] Mark Linton, Paul Calder, John Interrante, Steven Tang, and John Vlissides. InterViews Reference Manual. CSL, Stanford University, 3.1 edition, 1992.

[Lea88] Doug Lea. libg++, the GNU C++ library. In Proceedings of the 1988 USENIX C++ Conference, pages 243–256, Denver, CO, October 1988. USENIX Association.

[LG86] Barbara Liskov and John Guttag. Abstraction and Specification in Program Development. McGraw-Hill, New York, 1986.

[Lie85] Henry Lieberman. There's more to menu systems than meets the screen. In SIGGRAPH Computer Graphics, pages 181–189, San Francisco, CA, July 1985.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Object-Oriented Programming Systems, Langu-ages, and Applications Conference Proceedings, pages 214–223, Portland, OR, November 1986.

[Lin92] Mark A. Linton. Encapsulating a C++ library. In Proceedings of the 1992 USENIX C++ Conference, pages 57–66, Portland, OR, August 1992. ACM Press.

[LP93] Mark Linton and Chuck Price. Building distributed user interfaces with Fresco. In Proceedings of the 7th X Technical Conference, pages 77–87, Boston, MA, January 1993.

[LR93] Daniel C. Lynch and Marshall T. Rose. Internet System Handbook. Addison-Wesley, Reading, MA, 1993.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. Computer, 22(2):8–22, February 1989.

[Mar91] Bruce Martin. The separation of interface and implementation in C++. In Proceedings of the 1991 USENIX C++ Conference, pages 51–63, Washington, D.C., April 1991. USENIX Association.

[McC87] Paul McCullough. Transparent forwarding: First steps. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 331–341, Orlando, FL, October 1987. ACM Press.

[Mey88] Bertrand Meyer. Object-Oriented Software Construction. Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Mur93] Robert B. Murray. C++ Strategies and Tactics. Addison-Wesley, Reading, MA, 1993.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In SOOPPA Conference Proceedings, pages 145–161, Marist College, Poughkeepsie, NY, September 1990. ACM Press.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93), pages 66–73, Indianapolis, IN, February 1993.

[P+88] Andrew J. Palay et al. The Andrew Toolkit: An overview. In Proceedings of the 1988 Winter USENIX Technical Conference, pages 9–21, Dallas, TX, February 1988. USENIX Association.

[Par90] ParcPlace Systems, Mountain View, CA. ObjectWorks\Smalltalk Release 4 Users Guide, 1990.

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 341–346, Portland, OR, October 1986. ACM Press.

[Pug90] William Pugh. Skiplists: A probabilistic alternative to balanced trees. Communications of the ACM, 33(6):668–676, June 1990.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rum94] James Rumbaugh. The life of an object model: How the object model changes during development. Journal of Object-Oriented Programming, 7(1):24–32, March/April 1994.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, 10(5):595–609, September 1984.

[Sha90] Yen-Ping Shan. MoDE: A UIMS for Smalltalk. In ACM OOPSLA/ECOOP '90 Conference Proceedings, pages 258–268, Ottawa, Ontario, Canada, October 1990. ACM Press.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented languages. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 38–45, Portland, OR, November 1986. ACM Press.

[SS86] James C. Spohrer and Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? Communications of the ACM, 29(7):624–632, July 1986.

[SS94] Douglas C. Schmidt and Tatsuya Suda. The Service Configurator Framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons. In Proceeding of the Second International Workshop on Configurable Distributed Systems, pages 190–201, Pittsburgh, PA, March 1994. IEEE Computer Society.

[Str91] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, MA, 1991. Second Edition.

[Str93] Paul S. Strauss. IRIS Inventor, a 3D graphics toolkit. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 192–200, Washington, D.C., September 1993. ACM Press.

[Str94] Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley, Reading, MA, 1994.

[Sut63] I.E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. PhD thesis, MIT, 1963.

[Swe85] Richard E. Sweet. The Mesa programming environment. SIGPLAN Notices, 20(7):216–229, July 1985.

[Sym93a] Symantec Corporation, Cupertino, CA. Bedrock Developer's Architecture Kit, 1993.

[Sym93b] Symantec Corporation, Cupertino, CA. THINK Class Library Guide, 1993.

[Sza92] Duane Szafron. SPECTalk: An object-oriented data specification language. In Technology of Object-Oriented Languages and Systems (TOOLS 8), pages 123–138, Santa Barbara, CA, August 1992. Prentice Hall.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 227–242, Orlando, FL, October 1987. ACM Press.

[VL88] John M. Vlissides and Mark A. Linton. Applying object-oriented design to structured graphics. In Proceedings of the 1988 USENIX C++ Conference, pages 81–94, Denver, CO, October 1988. USENIX Association.

[VL90] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. ACM Transactions on Information Systems, 8(3):237–268, July 1990.

[WBJ90] Rebecca Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. Communications of the ACM, 33(9):104–124, 1990.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Designing Object-Oriented Software. Prentice Hall, Englewood Cliffs, NJ, 1990.

[WGM88] André Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 46–57, San Diego, CA, September 1988. ACM Press.

АЛФАВИТНЫЙ УКАЗАТЕЛЬ

И

К

М

Н

О

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: 03.2020. 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 18.03.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 36,120. Тираж 2000. Заказ 0000.

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*